LINEARIZABLE RELAXATIONS OF STACKS AND THEIR GENERALIZATIONS TO ORDERED DATA STRUCTURES

A Thesis
Presented to
The Academic Faculty

By

Erick Lin

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Computer Science with the
Research Option in the
College of Computing

Georgia Institute of Technology

May 2017

LINEARIZABLE RELAXATIONS OF STACKS AND THEIR GENERALIZATIONS TO ORDERED DATA STRUCTURES

Approved by:

Dr. Byron Boots, Advisor
School of Interactive Computing
Georgia Institute of Technology

Dr. Umakishore Ramachandran
School of Computer Science
Georgia Institute of Technology

Dr. Richard Peng
School of Computer Science
Georgia Institute of Technology

Date Approved: April 26, 2017

ACKNOWLEDGEMENTS

This work was supported in part by NSF grant 1526725. I am thankful for the guidance and mentorship provided by Edward Talmage and Prof. Jennifer Welch over the course of this research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Relaxation of semantics is a technique for improving the amortized distributed time complexity of data structures. We exhibit the first known algorithms implementing linearizable relaxed stacks in a partially synchronous, message-passing system, and proceed to show that relaxed priority queues reduce to relaxed stacks, meaning that their implementations are equally as fast in terms of amortized performance. Furthermore, restricting these new algorithms to relaxed queues improves on the previously best known upper bounds.

CHAPTER 1
INTRODUCTION AND BACKGROUND

Distributed computing infrastructures have played an increasingly large role in business, government, and scientific research in recent years, and reliance on these infrastructures, which include grids and clouds, will only continue to grow. For instance, the mission of the Worldwide LHC Computing Grid according to its website (wlcg-public.web.cern.ch) is "to provide computing resources to store, distribute, and analy[z]e the data generated by the Large Hadron Collider (LHC), making the data equally available to all partners, regardless of their physical location." Perhaps a more immediate example of a sizable collection of distributed systems is the Internet itself, whose users now include over half of the world's population and nearly ninety percent in North America. From the perspective of researchers, the growing complexity of these infrastructures may be addressed by breakthroughs in theoretical knowledge regarding their structure and verification.

One of the goals of distributed computing is to ensure that the participating processes always have the most up-to-date information regarding the system of interest. In a *message-passing system*, each process is unaware of any other's actions unless the other communicates to it directly, but the illusion of a shared state must still be maintained in order to derive any benefit over a non-distributed system. For example, operating systems such as Windows and Linux support remote procedure call protocols which allow a procedure to execute in a separate address space while appearing as a normal procedure call to the user, helping to expedite the development of client-server applications. It is crucial in these situations that each party is guaranteed to have non-contradicting knowledge of shared information while minimizing the amount of time spent on interprocess communication.

To organize informational interactions, we often use *ordered data types* which include *queues*, *stacks*, and *priority queues*. These describe sets of data in which elements can be

added, retrieved, or removed one at a time by *operations* called by separate processes, but differ in the order in which elements are retrieved or removed. In queues, elements added first have the highest priority, while stacks follow the opposite order in which these are the elements added last; priority queues are a generalization of the first two in which elements are intrinsically given their own priorities according to any specific function.

In order to evaluate orderings on operation instances in concurrent executions based on the specification of the data structure, many different types of *consistency conditions* have been proposed. The strongest of these consistency conditions is *linearizability*, in which instances of operations called by processes can be ordered in the same manner in all processes, ensuring that the processes always finish in the same shared state [1].

Previous research on lower and upper bounds of strict queues and stacks has shown that *mixed accessor/mutator* operations such as $Dequeue$ and Pop must be slow in the worst case to satisfy linearizability, because the calling process is required to wait for all other processes to communicate their preceding operation instances [2]. More generally, linearizable data structures are costly to implement in both shared-memory [3] and message-passing systems [4], [5].

One workaround has been to replace linearizability with weaker consistency conditions, such as quasi-linearizability [6], sequential consistency [7], eventual consistency [8], quiescent consistency [9], and others, to yield improved performance [5]. The negation of the presence of the shared state has been the foremost drawback of these approaches, however.

More recently, attention has shifted to relaxing instead the *semantics* of operations [10], [6]. Examples of relaxations, in terms of a parameter k, include

- the *Out-of-Order* relaxation which allows $Dequeue$ to remove any of the k highest-priority elements (i.e., best candidates),
- the *Lateness* relaxation which allows up to $k-1$ $Dequeue$ instances to remove any element without restriction before the next $Dequeue$ instance must remove the *top-most* element, or single element of highest priority, and

- the *Restricted* relaxation, defined by the requirement of satisfying the first two.

[11] was the first to investigate lower and upper bounds of relaxed ordered data structures, and discovered that relaxation provided beneficial results for *amortized* time complexity. In particular, lower bounds were given for a comprehensive variety of relaxed queues and stacks, and upper bounds were provided for the Out-of-Order k-Relaxed Queue and Restricted k-Relaxed Queue.

At the outset, we conceive of a number of different types of *relaxed stacks* (Section 2). Then we introduce the extensive theory leading to an algorithm implementing an Out-of-Order k-Relaxed Stack (Section 3), forming the basis for algorithms implementing several other types of "stronger" relaxed stacks (Appendix A).

Section 4 focuses on further extensions of these results. The latter phase of our paper consists of prescribing an algorithmic reduction from *relaxed priority queue* implementations to those of relaxed stacks. Finally, we come around full circle by restricting relaxed priority queues to *relaxed queues* which, when combined with optimizations tailored for the new techniques, improve on the previously discovered upper bounds.

CHAPTER 2

DEFINITIONS AND MODEL

We begin by defining abstract data types in the sequential setting, in the same manner as described in [11].

A data type T of an object provides a set of operations $OPS(T)$ defined on this type. Each operation OP in $OPS(T)$ has an invocation OP_{inv}, which may include an argument, and a response OP_{resp}, which may include a return value. Any invocation $OP_{inv}(arg)$ and its immediately following response $OP_{resp}(ret)$ are indivisible in the sequential case, so the pair can be written as $OP(arg, ret)$, which is called an *instance* or *operation instance* of OP.

The semantics of a data type T are defined by a set of operation instances $L(T)$, called the *legal sequences*, which must satisfy the following properties:

- *Prefix Closure*: If ρ is in $L(T)$, then every prefix of ρ, including the empty sequence, is in $L(T)$.
- *Completeness*: If ρ is in $L(T)$, then for every argument arg and every operation invocation OP_{inv}, there exists a response $OP_{resp}(ret)$ for $OP_{inv}(arg)$ such that $\rho \cdot OP(arg, ret)$ is in $L(T)$.

2.1 Relaxed Stacks

We initially define the strict or *unrelaxed* Stack data structure as a baseline.

Definition 1. *A* Stack *over a set of values V is a data type with two operations:*

- $Push(val, -), val \in V$,
- $Pop(-, val), val \in V \cup \{\perp\}$.

Given a sequence ρ of operation instances on a Stack, if we define the following terms:

- *A $Push(val, -)$ is* matched *if there exists a $Pop(-, val)$ in ρ. If this is the case, $Push(val, -)$ is said to* match *$Pop(-, val)$ in ρ.*
- *A $Pop(-, val)$ in ρ is* headmost *if there exists a $Push(val, -)$ in ρ such that every $Push(val', -)$ succeeding $Push(val, -)$ in ρ is matched by a $Pop(-, val')$ appearing in ρ before $Pop(-, val)$.*

then ρ is legal iff it satisfies the following conditions:

(C1) The empty sequence is legal.

(C2) If ρ is a legal sequence of operation instances, then $\rho \cdot Push(val, -)$ is legal iff there is no $Push(val, -)$ already in ρ [†].

(C3) If ρ is a legal sequence of operation instances, then $\rho \cdot Pop(-, val)$, $val \neq \perp$ is legal iff $Pop(-, val)$ is headmost.

$\rho \cdot Pop(-, \perp)$ is legal iff every instance of $Push$ in ρ is matched.

In particular, $Pop(-, \perp)$ is always a Pop that is not headmost.

We now describe a number of different ways to relax the Stack data structure, each of which is defined in terms of a *relaxation parameter* k specifying the degree of relaxation. To clarify, the Pop operation is what will be relaxed in particular.

Intuitively, the following definition describes the relaxation in which a Pop may remove any of the top k elements.

Definition 2. *An* Out-of-Order k-Relaxed Stack *over a set of values V provides the same operations as in Definition 1.*

A sequence ρ of operation instances on an Out-of-Order k-Relaxed Stack is legal iff it satisfies conditions (C1) and (C2) from Definition 1, as well as

[†]Unique values can easily be achieved by timestamping elements as they are added.

(C3) If ρ is a legal sequence of operation instances, then $\rho \cdot Pop(-, val), val \neq \perp$ is legal iff there is an unmatched $Push(val, -)$ in ρ that is succeeded by fewer than k unmatched $Push$ instances in ρ.

Furthermore, $\rho \cdot Pop(-, \perp)$ is legal iff there are fewer than k unmatched $Push$ instances in ρ.

In the definition that immediately follows, successive Pop instances may remove elements from anywhere in the stack (or even $\perp$) up to $k - 1$ times before being required to remove the topmost element.

Definition 3. *A* Lateness k-Relaxed Stack *over a set of values V provides the same operations as in Definition 1.*

Given a sequence ρ of operation instances on a Lateness k-Relaxed Stack, if we define the following additional terms:

- *A $Pop(-, val)$ in ρ is* resolved *if $Pop(-, val)$ is either headmost or succeeded by a headmost Pop instance.*

  A value val is resolved *in ρ if there exists a $Pop(-, val)$ in ρ that is resolved.*

  A matched $Push(val, -)$ in ρ is resolved *if some $Pop(-, val)$ it matches is resolved.*

- *The* lateness *of ρ, $\lambda(\rho)$, is the number of unresolved Pop instances in ρ.*

then ρ is legal iff it satisfies conditions (C1) and (C2) from Definition 1, as well as

(C3) If ρ is a legal sequence of operation instances, then $\rho \cdot Pop(-, val)$ is legal iff there is an unmatched $Push(val, -)$ in ρ and $\lambda(\rho \cdot Pop(-, val)) < k$.

Furthermore, $\rho \cdot Pop(-, \perp)$ is legal iff every instance of $Push$ in ρ is matched or $\lambda(\rho \cdot Pop(-, \perp)) < k$.

The next definition is identical except that lateness is replaced by a new concept called current-top lateness. This time the intent is to keep track of a separate lateness counter for each element, which comes into effect when it is the topmost element.

Definition 4. *A* Per-Top Lateness k-Relaxed Stack *over a set of values V provides the same operations as in Definition 1.*

Given a sequence ρ of operation instances on a Per-Top Lateness k-Relaxed Stack, if we define the following additional terms:

- *A $Pop(-, val)$ in ρ is* own-top resolved *if $Pop(-, val)$ is either headmost or succeeded by a Pop instance whose matching $Push$ instance is the last $Push$ instance preceding $Pop(-, val)$.*

  A value val is own-top resolved *in ρ if there exists a $Pop(-, val)$ in ρ that is own-top resolved.*

  A matched $Push(val, -)$ in ρ is own-top resolved *if some $Pop(-, val)$ it matches is own-top resolved.*

- *The* current-top lateness *of ρ, $\gamma(\rho)$, is the number of own-top unresolved Pop instances succeeding the last unmatched $Push$ in ρ.*

then ρ is legal iff it satisfies conditions (C1) and (C2) from Definition 1, as well as

(C3) If ρ is a legal sequence of operation instances, then $\rho \cdot Pop(-, val), val \neq \perp$ is legal iff there is an unmatched $Push(val, -)$ in ρ and $\gamma(\rho \cdot Pop(-, val)) < k$.

Furthermore, $\rho \cdot Pop(-, \perp)$ is legal iff every instance of $Push$ in ρ is matched or $\gamma(\rho \cdot Pop(-, \perp)) < k$.

We also present two types of relaxed stacks which combine the out-of-order and lateness restrictions.

Definition 5. *A* Restricted k-Relaxed Stack *over a set of values V provides the same operations as in Definition 3.*

A sequence ρ of operation instances on a Restricted k-Relaxed Stack is legal iff it satisfies conditions (C1) and (C2) from Definition 1, as well as

(C3) If ρ is a legal sequence of operation instances, then $\rho \cdot Pop(-, val)$, $val \neq \perp$ is legal iff there is an unmatched $Push(val, -)$ in ρ that is succeeded by fewer than $k - \lambda(\rho \cdot Pop(-, val))$ unmatched $Pushes$ in ρ.

Furthermore, $\rho \cdot Pop(-, \perp)$ is legal iff there are fewer than $k - \lambda(\rho \cdot Pop(-, \perp))$ unmatched $Pushes$ succeeding the last unmatched $Push$.

Definition 6. *A* Windowed k-Relaxed Stack *over a set of values V provides the same operations as in Definition 3.*

A sequence ρ of operation instances on a Windowed k-Relaxed Stack is legal iff it satisfies conditions (C1) and (C2) from Definition 1, as well as

(C3) If ρ is a legal sequence of operation instances, then $\rho \cdot Pop(-, val)$, $val \neq \perp$ is legal iff there is an unmatched $Push(val, -)$ in ρ that is succeeded by fewer than k $Pushes$ that are either unmatched or unresolved in ρ.

Furthermore, $\rho \cdot Pop(-, \perp)$ is legal iff there are fewer than k $Push$ instances that are either unmatched or unresolved.

Of the last two definitions, the first states that the top "k minus lateness" elements are the set of elements legal for a Pop to return. The second states that the legal set of elements that can be Popped includes all present elements in a *window* of (constant) size k from the top, where windows include all present and unresolved elements; an informal description is that $Push$ing an element "shifts the window up" by 1, while Popping the topmost element resolves all the unresolved elements so that they disappear from the window, and then "shifts the window down" until it is aligned with the topmost element present in the stack.

We can also state the Per-Top versions of Definitions 5 and 6 using current-top lateness and own-top unresolved elements just as Definition 4 does for Definition 3. What is interesting to consider in the case of a Per-Top Windowed k-Relaxed Stack is that the own-top unresolved elements are *invisible*, or not contained in the window, when the element whose

removal would resolve them is not currently the top of the stack, but they become visible again if that element returns to the top.

Our final definition allows Pop instances to leave the stack unchanged (in other words, to act as instances that perform a peek operation) up to $k-1$ times for each element.

Definition 7. *A* Stuttering k-Relaxed Stack *over a set of values V provides the same operations as in Definition 1.*

Given a sequence ρ of operation instances on a Stuttering k-Relaxed Stack, if we define the following additional terms:

- *The* stutter count *of a $Pop(-, val), val \neq \perp$ is the number of times $Pop(-, val)$ appears in ρ.*

  The stutter count *of a value val, $s(val)$, is the stutter count of $Pop(-, val)$ in ρ.*

- *The* stutter count *of ρ, $s(\rho)$, is the stutter count of the last Pop instance in ρ.*

- *A $Pop(-, val), val \neq \perp$ in ρ is* removed *if $s(val) = k$ or $Pop(-, val)$ is succeeded by a $Pop(-, val')$ where $val' \neq val$.*

  A value val is removed *in ρ if there exists a $Pop(-, val)$ in ρ that is removed.*

then ρ is legal iff it satisfies conditions (C1) and (C2) from Definition 1, as well as

(C3) If ρ is a legal sequence, then $\rho \cdot Pop(-, val)$, $val \neq \perp$ is legal iff

- *there is a $Push(val, -)$ in ρ,*
- *every $Push$ succeeding $Push(val, -)$ is matched, and*
- *val is not removed in ρ.*

$\rho \cdot Pop(-, \perp)$ is legal iff every $Push$ in ρ is matched.

2.2 System Model

The distributed system model we adopt is defined in [11], which we re-state in this section. We consider a set $\Pi = \{p_0, \cdots, p_{n-1}\}$ of n processes, each modeled as a state machine

whose transitions are triggered by occurrences of three kinds of events: the invocation of an operation instance, the receipt of a message, and the expiration of a timer.

A *step* of a process is a 6-tuple (s, T, C, M, R, s'), where s and s' are respectively the old and new states, T is a trigger event, C is a local clock value taken from the reals, M is the set of messages sent, and R is either empty or the response of an operation instance, such that M, R, and s' are the result of the transition function operation on s, T, and C.

A *view* of a process is a sequence of steps such that

- the old state of the first step is an initial state;
- the old state of each subsequent step is the same as the new state of the step following that one;
- the value of each timer in the old state of each step is at most the clock time of the step;
- if the trigger for a step is a timer going off, then there exists a timer in the old state of the step whose value equals the clock time of the step;
- the clock times in steps are increasing, and if the sequence is infinite then they are unbounded;
- at most one operation instance is pending at a time.

A *timed view* is a view in which a real number, or the *real time*, is associated with each step. The real time must proceed at the same rate as the clock time; i.e., there must exist a real number c, called the *clock offset*, such that for every step, the difference between the clock time and the real time is exactly c.

A *run* is a set of n timed views, one for each process, such that every message receipt has exactly one matching message send, and every message send has at most one message receipt. A run is *complete* if

- every message sent is received; and

- each timed view is either infinite or ends in a state in which no timers are set.

A run is *admissible* with respect to parameters ϵ, d, and u if

- the local clocks are *synchronized* to within ϵ time; i.e., if c_i is the local clock offset for process p_i, then $|c_i - c_j| \leq \epsilon$ for all processes p_i and p_j, and
- the delays for received messages lie within the range $[d - u, d]$, and if time t has the send of a message with no matching receive, then the last real time for any step by the intended recipient process must be less than $t + d$.

Implicit in these requirements is that $\epsilon < u < d$, and in fact, we know that $\epsilon < (1 - 1/n)u$ [12]. We further assume that any message from a process to itself is simulated as taking the minimum message delay $d - u$.

We will consider only algorithms which are *eventually quiescent* – that every complete, admissible run with a finite number of operation instances is finite, implying every view is finite.

2.3 Correctness Condition

Our objective is to provide linearizable implementations of data types described in Section 2.1 that have their basis in the message-passing model described in Section 2.2. These algorithms must satisfy the following conditions:

- *Liveness*: In every complete, admissible run, the matchings between operation invocations and responses exhibit a one-to-one correspondence. This allows matching invocations and responses to be paired up as operation instances.
- *Linearizability*: For every complete, admissible run R, there is a permutation π of the set of operation instances in R such that (i) π is legal and (ii) with respect to real time in R, if operation instance op_1 responds before operation instance op_2 is invoked, then op_1 precedes op_2 in π. π is called a *linearization* of R.

2.4 Performance Metrics

Lastly, we give two upper bound metrics on the performance of data types in this model.

The *worst-case time complexity* of operation OP is the maximum over every instance of OP that may occur in any complete, admissible run of the real time that elapses between the invocation of the instance and its response.

The *amortized time complexity* of operation OP is the least upper bound over every complete, admissible run R and every real time t of $avg\_time(R, OP, t)$, where $avg\_time(R, OP, t)$ is the sum of the elapsed time taken by all instances of OP in R which complete by time t divided by the number of such instances of OP.

CHAPTER 3
ALGORITHM AND UPPER BOUNDS

In this paper, we contribute implementations of several kinds of relaxed stacks (Out-of-Order in this section, with the remaining deferred to the Appendix) according to the specified model. Each process instantiates a local copy of the relaxed data structure that is referred to as its *local stack*, and synchronization of these local stacks is a cornerstone of the reasoning involved. The algorithms are interrupt-driven, with three types of events that may trigger actions at a process: the invocation of an operation instance, the receipt of a message from another process, and the expiration of a earlier-set timer. The timers ensure that event occurrences are coordinated despite the delays in communication between processes, and each is associated with an operation instance and an action which are used by the algorithm to determine the return value of the operation instance.

In the algorithms, the PUSH or POP event handler takes place in a process p_i when $Push$ or Pop instances respectively are invoked in p_i, and a **return** statement refers to the response accompanied by a return value (or ACK, if there is no return value) of the previous operation instance invoked in the same process. Each operation instance causes the EXECUTELOCALLY function to be executed at all processes a given amount time after the operation instance is invoked, and in each process this is referred to as the instance's *execution* or *local execution*.

We use *timestamps* to determine the order in which operation instances should be executed that satisfies the linearizability condition. Each operation instance upon invocation is given a timestamp, an ordered pair consisting of the local clock time and the id of the process in which the instance was invoked. We maintain at each process a priority queue of operation instances waiting for the correct time to execute locally, the priority function of which is the lowest-first lexicographic ordering of timestamps. If the timer for execut-

ing locally expires earlier in some operation instance with a later timestamp compared to another, then the operation with the earlier timestamp is allowed to execute locally first, canceling its own timer.

We assume that all built-in functions used in the pseudocode, such as $\min$, are deterministic, i.e., there is some method of tie-breaking that guarantees a unique output for every input. In this way, different processes that execute the same pseudocode will get the same result.

3.0.1 Parameters

The parameters we use which are universal to the rest of this paper include

- $n \geq 1$, the number of processes;
- $k \geq 0$, the relaxation parameter;
- $l := \lfloor k/n \rfloor$;
- $l_s := \lceil k/n \rceil - 1$; and
- $\tau \in [1, l]$, the trade-off parameter (we briefly introduce it here by giving the guideline that a higher value of τ favors the performance of $Push$es over Pops);

all of which are integer-valued.

3.0.2 Local Variables

The algorithms use a number of local variables, which are specified completely here (it is recommended that this subsection be referred back to for reference):

- $localTime$: Current local time
- $lStack$: Set representation of local stack, initially empty. Elements have the following associated fields:

- $label$, which takes on process ids, $dummy$, and $null$ (signifying that the element is unlabeled) as values and is initially $null$
- $popped$, a Boolean field which is initially $false$
- $t_{volatile}$, a time which is initially $-\infty$
- $higher$, which takes on elements and $null$ as values and is initially $null$
- $lower$, which takes on elements and $null$ as values and is initially $null$
- $canPopByLabel$, a Boolean field which is initially $true$
- $unresolved$, a Boolean field which is initially $false$
- t_{popped}, a time which is initially ∞; used in Algorithm 6 only

Operations on $lStack$ include:

- $push(val)$: inserts a new element val at the top
- $pop()$: removes and returns the top non-$popped$ element
- $peek()$: returns the top non-$popped$ element
- $popByLabel(p_j)$: removes and returns the topmost non-$popped$, $canPopByLabel$ element labeled p_j, or returns $\perp$ if none exists
- $peekByLabel(p_j)$: returns the topmost non-$popped$, $canPopByLabel$ element labeled p_j, or $\perp$ if none exists
- $peekByLabel(p_j, S)$: returns the topmost non-$popped$, $canPopByLabel$ element labeled p_j which is also in the set S, or $\perp$ if none exists
- $topAvailableByLabel(p_j)$: returns the topmost non-$popped$ element labeled p_j, or $\perp$ if none exists
- $peekBySet(S)$: returns the topmost element which is also in the set S, or $\perp$ if none exists
- $getFromTop(m)$: returns, without removing, the mth-topmost element, or $\perp$ if $size() < m$ or $m \leq 0$
- $setByLabel(p_j)$: returns the set of all non-$popped$ elements labeled p_j

  - $topSet(m)$: returns the set of the $\min\{m, lStack.size()\}$ topmost elements in $lStack$
  - $topSetBySet(m, S)$: returns the set of the $\min\{m, lStack.size(), |S|\}$ topmost elements in $lStack$ which are also in the set S
  - $contains(val)$: returns $true$ if val is an element in $lStack$, $false$ otherwise
  - $size()$: returns current number of elements
  - $labeledSize()$: returns current number of labeled elements
  - $remove(val)$: removes the element sharing the same value as val
  - $label(p_j, val)$: sets the label of the element val to p_j and returns the process id number of its previous label if $val \neq \perp$
- $Pending$: Priority queue containing operation instances keyed by timestamp; initially empty; supports standard operations $insert(val, ts)$, $min()$, $extractMin()$
- $localFastPushesActive$: Integer-valued variable, initially 0
- $popsInPending$: Integer-valued variable, initially 0
- $reservedQueues[]$: Array of size n of queues of data elements, each initially empty; support standard operations $enqueue(val)$, $dequeue()$ and $peek()$ (the former of which does nothing if $val = \perp$, and the latter two of which return $\perp$ if it is empty)
- $localPopsInvoked$: Integer-valued variable, initially 0
- $localPopsExecuted$: Integer-valued variable, initially 0
- $kMinusLateness$: Integer-valued variable, initially k

Here we define modifications of τ which depend on the size of the local stack.

Definition 8. *At each process p_i, the* packing-τ, τ_p, *and the* covering-τ, τ_c, *are defined respectively as*

$$\tau_p := \min\left\{\tau, \max\left\{\left\lceil\frac{lStack.size() - k}{n}\right\rceil + \tau, 0\right\}\right\}$$

and

$$\tau_c := \max\left\{\tau, \left\lceil \frac{k - lStack.size()}{n} \right\rceil\right\}.$$

Definition 8 says that $\tau_p \leq \tau$, with equality if and only if the local stack is of size k or greater; otherwise, it gives approximately the difference between the size of the local stack and $k - \tau n$, divided by the number of processes n.

Likewise, $\tau_c \geq \tau$, with equality if and only if the local stack is of size $k - \tau n$ or greater; otherwise, it gives approximately the difference between k and the size of the local stack, divided by n.

In the algorithms given throughout the rest of this section, it is assumed that $k \geq n$. All three of them give improved amortized performance over algorithms for unrelaxed Stacks, increasing as k increases by multiples of n.

3.1 Out-of-Order Relaxed Stacks

Algorithm 1 gives a minimal implementation of the Out-of-Order k-Relaxed Stack (Definition 2) for the purpose of understanding, whereas Algorithm 2 gives an improved implementation which will be the one studied exhaustively for correctness and performance.

3.1.1 Minimal Implementation

The initial motivating idea is that in each process' local stack, all elements legal to return by locally executing Pops are *labeled* with process ids, signifying that the element may only be returned by that process. This idea allows for elements labeled with the id of a process at the invocation of a Pop in that process to be *claimed* by that Pop, meaning that the Pop returns that element's value in the earliest time possible, ϵ after invocation. Such a Pop is called a *fast* Pop, an operation abbreviated as pop_f in the algorithm (This naturally leads to any other Pop being called a *slow* Pop, abbreviated pop_s.).

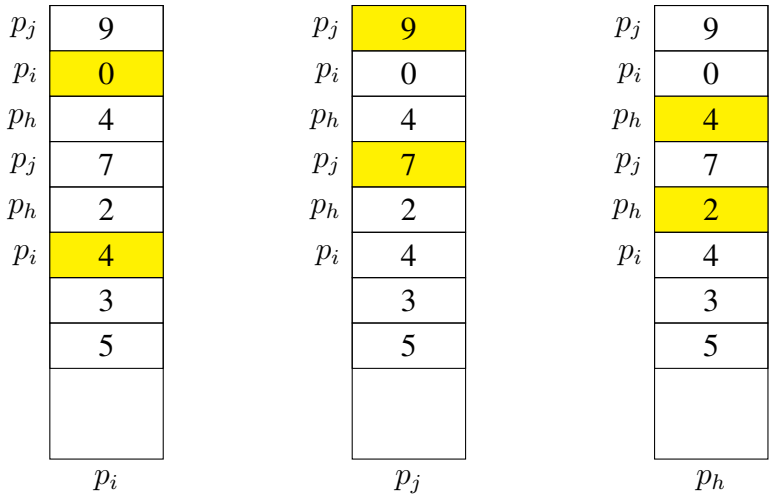

Figure 3.1: Partitioning elements among the processes using labels, with $k = 6, n = 3$.

If all elements labeled with the id in the local stack are already fast Popped at the time another Pop is invoked by a process, then that Pop is a slow Pop and waits until another element is labeled with the required process id, which we guarantee occurs before the Pop executes locally in the same process.

When the size of each local stack is large enough, a *uniformity invariant* ensures that a certain number of elements are labeled with each process id and can be claimed by fast Pops, which leads to an amortized performance improvement for this relaxed stack, contrasting with unrelaxed stacks.

A sample run involving processes p_i, p_j, and p_h is depicted in the figure on the following page, where $\rightarrow$ is the forward direction in time. As we can see, the invocation of new operation instances is permitted in a process as long as all previous ones have responded. (Omitted from the figure for the sake of understanding are the clock offsets between processes, as well as the fact that after each message is received, the operation instance sits in $Pending$ for a specified amount of time before executing locally.)

The top k elements are always labeled in each local stack. It is important to note that not all labeled elements are necessarily legal to return by fast Pops. We discuss the restrictions on elements that can be returned by fast Pops in the context of active operation instances.

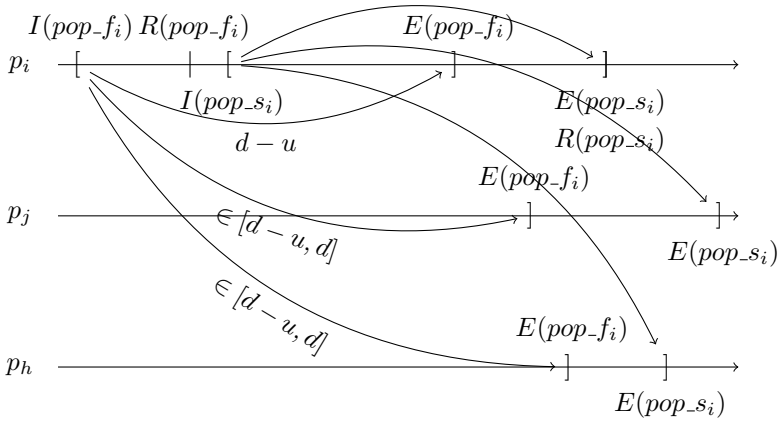

Figure 3.2: Invocation (I), response (R), and local execution (E) of Pop instances.

Definition 9. *An operation instance is* active *in a process p_i if it has been invoked (in terms of real time) but not yet executed locally in p_i.*

Definition 10. *Two operation instances* mutually active *if, in the process in which the one with the later timestamp is invoked, the one with the earlier timestamp has not yet completed local execution by the other's invocation time.*

One example scenario is that the kth-topmost element in the local stack is illegal to return by a Pop if there is a mutually active $Push$ and no mutually active Pops, because since the $Push$ has an earlier timestamp, the number of unmatched $Push$es preceding the Pop in the linearized order of operation instances is here not less than k.

Furthermore, there may be more than a single mutually active $Push$, as the maximum number among all processes is n. This means that at the invocation of a Pop, which we call pop, in process p_i when the local stack has at least $k - n$ elements, there might be up to n $Push$es mutually active with pop that precede pop in the linearization. We conclude that only $k - n$ elements in p_i's local stack are guaranteed legal to return by pop (if they have not already been returned by other fast Pops).

A silver lining is that other Pops mutually active with pop that precede pop in the linearization increase the number of elements legal to return by pop, because they decrease the number of unmatched $Push$es in the sequence. When a process receives the message sent

at most d time after the real time of invocation of a Pop, it increments its $popsInPending$ counter. As expected, each Pop decreases the $popsInPending$ counter after it executes locally. Now we have that at least the topmost $k - n + popsInPending$ elements in p_i's local stack at pop's invocation are legal for pop to return, if they have not already been returned by other fast Pops.

Here we introduce the notion of the *safe region*, which is a set of $\min\{k - n, lStack.size()\}$ elements in the local stack legal for fast Pops to return. The set of labeled elements outside the safe region is referred to as the *contention region*. The uniformity invariant introduced earlier applies to the safe region, and intuitively describes the even distribution of labels among elements in the safe region if it is of size $k - n$. Preserving the uniformity invariant at all processes is one reason why a fast Pop marks an element $popped$ instead of directly removing it from the local stack during invocation (the other motive being that the deterministically-chosen new label at line 47 is the same in the local execution at every process). Recall that $peekByLabel()$ and $popByLabel()$ return only elements that are not $popped$.

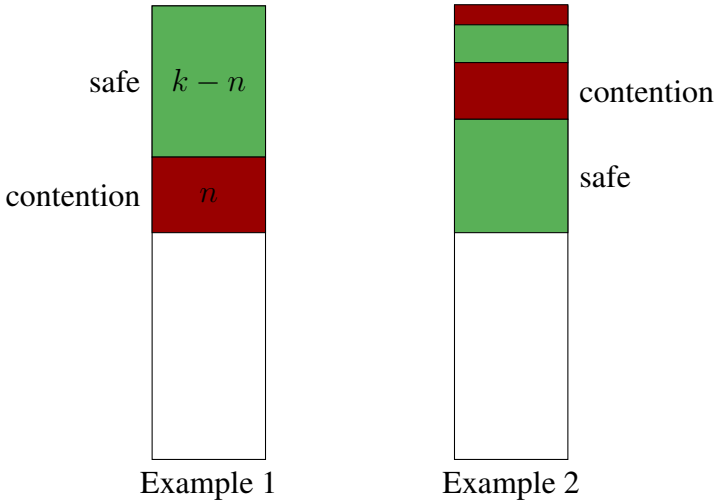

Figure 3.3: The safe and contention regions.

Maintaining the uniformity invariant requires that whenever the local execution of a Pop in p_i removes an element from the safe region and an element previously not in the safe

region enters it, it must *relabel* some element in the safe region. By default, the relabeled element is the newly entering element, which is also the bottommost in the updated safe region. However, one issue this causes is that the element could have already been returned by another Pop invoked in p_i, which we call pop, so it is dangerous to relabel it p_j, as the element is not $popped$ in the view of p_j and could be returned again by a fast Pop invoked in p_j, causing an illegal linearization of operation instances since pop already matches the $Push$ that added this element.

To decide which element should be relabeled, we give the following procedure. The safe region is initially the topmost $k - n$ elements in the local stack. Any new element added by the local execution of a $Push$ in an any process p_h is temporarily marked *volatile*, meaning it can be relabeled, for as long as required by a Pop invoked (in any process) before the local execution of the $Push$ (in that process) to itself locally execute in p_h, removing its returned element from p_h's local stack. pop from the previous paragraph is such a Pop, because the only way for an element in the initial safe region to leave the topmost $k - n$ elements is through locally executing $Push$es. Thus, if another locally executing Pop intervenes by removing some element in the safe region from the local stack, it relabels the topmost volatile element instead of the element returned by pop.

It is illegal to return volatile elements that are about to be relabeled. We know whether they will be by Pops that were invoked at least d time ago, because they contribute to the $popsInPending$ counter; therefore, the topmost $popsInPending$ volatile elements in the local stack are excluded from the safe region. We also guarantee that any volatile elements outside the topmost $popsInPending$ are legal to return by an invoking Pop, because this Pop sends a message to all processes to mark its claimed element non-volatile within d time. Any Pops not contributing to $popsInPending$ will not execute locally in less than d time, because in this algorithm operation instances execute locally in at least $2d$ time.

We add that whenever a locally executing $Push$ adds a new element to the local stack, which is then marked volatile, it is labeled with the same label as the element that was

moved out of the topmost $k - n$ in the local stack. These two elements form a construct called a $(higher, lower)$-*pair*, which offers a number of guarantees. First of all, the topmost $popsInPending$ elements that are the $lower$ in some $(higher, lower)$-pair are included in the safe region, which allows the size of the safe region to stay at size $k - n$. We show that in this way, the local executions of $Push$es also preserve the uniformity invariant.

At this point, we are able to state the *labeling invariant* that the set of labeled elements is not only the elements legal for a locally executing Pop to return, but also elements which will become legal to return when all Pops contributing to $popsInPending$ are executed locally. We observe that if any elements are labeled, say a number m, then they must be the top m elements in the local stack.

The algorithm we have described so far is a complete implementation the Out-of-Order k-Relaxed Stack, and we give this implementation in Algorithm 1.

Algorithm 1 A minimal version of the pseudocode for each process p_i implementing an Out-of-Order k-Relaxed Stack, where $k \geq n$.

```
1:  HandleEvent PUSH(val)
2:      send (push_s, val, ⟨localTime, i⟩) to all
3:  HandleEvent POP
4:      if lStack.peekByLabel(p_i, safeRegion()) ≠ ⊥ or lStack.size() < k − n then
5:          let ret := lStack.peekByLabel(p_i, safeRegion())
6:          ret.popped := true
7:          send (pop_f, ret, ⟨localTime, i⟩) to all
8:          setTimer(ϵ, ⟨pop_f, ret, null⟩, respond)
9:      else send (pop_s, null, ⟨localTime, i⟩) to all
10: HandleEvent RECEIVE (op, val, ts) FROM p_j
11:     Pending.insert(⟨op, val, ts⟩)
12:     setTimer(d + u, ⟨op, val, ts⟩, execute)
13:     if op == pop_f or op == pop_s then
14:         popsInPending++
15:     if op == pop_f then
16:         resetVolatility(val)
17: HandleEvent EXPIRETIMER(⟨op, val, ⟨∗, j⟩⟩, respond)
18:     if op == pop_f then return val
19:     else return ACK
```

20: **HandleEvent** EXPIRETIMER($\langle op, val, ts\rangle, execute$)

21: **while** $ts \geq Pending.min()$ **do**

22: $\langle op', val', ts'\rangle := Pending.extractMin()$

23: $executeLocally(op', val', ts')$

24: $cancelTimer(\langle op', val', ts'\rangle, execute)$

25: **function** EXECUTELOCALLY($op, val, \langle time, j\rangle$)

26: **if** $op == push\_f$ or $op == push\_s$ **then**

27: $lStack.push(val)$

28: **if** $lStack.size() \leq k - n$ **then**

29: $lStack.label(\text{argmin}_{p_h} |lStack.setByLabel(p_h) \cap safeRegion()|, val)$

30: **else**

31: $lStack.label(lStack.getFromTop(k - n + 1).label, val)$

32: $setVolatility(val, lStack.getFromTop(k - n + 1))$

33: $lStack.label(null, lStack.getFromTop(k +$
$\min\{|lowerRegion()|, popsInPending\} + 1)$

34: **else**

35: **if** $op == pop\_f$ **then** let $toRemove := val$

36: **else**

37: let $toRemove := lStack.peekByLabel(p_i)$

38: **return** $toRemove$

39: **if** $toRemove.higher \neq null$ **then** $resetVolatility(toRemove.higher)$

40: $popsInPending--$

41: **if** $toRemove \in safeRegion()$ **then**

42: **if** $volatileRegion() \cap lStack.topSet(k - n) \neq \emptyset$ **then**

43: let $toRelabel := lStack.peekBySet(volatileRegion())$

44: $resetVolatility(toRelabel)$

45: **else** let $toRelabel := lStack.getFromTop(k - n + 1)$

46: $lStack.label(j, toRelabel)$

47: $lStack.remove(toRemove)$

48: **if** $lStack.labeledSize() < k$ **then** $lStack.label(dummy, lStack.peekByLabel(null))$

49: **function** SAFEREGION

50: **return** $[topSet(k - n) \cup topSetBySet(popsInPending, lowerRegion())] \setminus$
$topSetBySet(popsInPending, volatileRegion())$

51: **function** VOLATILEREGION

52: **return** $\{elem \in lStack.topSet(k - n) : elem.t_{volatile} > localTime - (2d + 2u + \epsilon)\}$

53: **function** LOWERREGION

54: **return** $\{elem.lower : elem \in volatileRegion()\}$

55: **function** SETVOLATILITY($elem, other$)

56: $elem.t_{volatile} := localTime$

57: $elem.lower := other$

58: $elem.lower.higher := elem$

59: **function** RESETVOLATILITY($elem$)

60: $elem.t_{volatile} := -\infty$

61: $elem.lower.higher := null$

62: $elem.lower := null$

3.1.2 Improvements

Instead of allowing only a single active $Push$ at a time per process, we may permit up to τ, so that the maximum number of mutually active $Push$es among all the processes is τn. In the same spirit as before, $Push$es deciding at invocation to respond in ϵ time are referred to as *fast $Push$es* (abbreviated $push\_f$), and other $Push$es are referred to as *slow $Push$es* (abbreviated $push\_s$).

The trade-off inherent in the τ parameter is that the size of the safe region is now $k-\tau n$, and all occurrences of $k - n$ in Algorithm 1 are now replaced by $k - \tau n$ in Algorithm 2. Hence, another interpretation of τ is the size of the contention region divided by n when k elements are labeled.

We generalize our usage of τ to τ_c and τ_p as follows. For a local stack of any size, not just $k - \tau n$ or greater, the maximum number of mutually active $Push$es is actually permitted to be $\tau_c n$. τ_p gives the size of the contention region divided by the number of processes when fewer than k elements are labeled.

Lastly, we describe an optimization on operation response time. Given that the safe region is of size $k - \tau n$, the idea is that if a series of $l - \tau$ fast Pops is invoked and respond in p_i as quickly as possible in a process before a slow Pop is invoked, then the slow Pop only needs to wait until the first fast Pop has executed locally, labeling another element in the local stack p_i, before the slow Pop may return the element newly labeled p_i. We use an array of *reserved queues*, one dedicated to Pops invoked by each process, whose purpose

is to guarantee that the element returned by a Pop is the same as the element removed from the local stack by its local execution in each process. If the size of the safe region is less than $k - \tau n$, it is possible that fewer than $l - \tau$ elements in the safe region are labeled p_i, but it is in this case legal by the uniformity invariant for fast Pops to return $\perp$.

A similar optimization is made for slow $Push$es, this time accomplished by limiting the number of active fast $Push$es in a process to $\tau_c - 1$. We dub this overall optimization technique "short-circuiting," with the slow $Push$es and Pops referred to as "short-circuited".

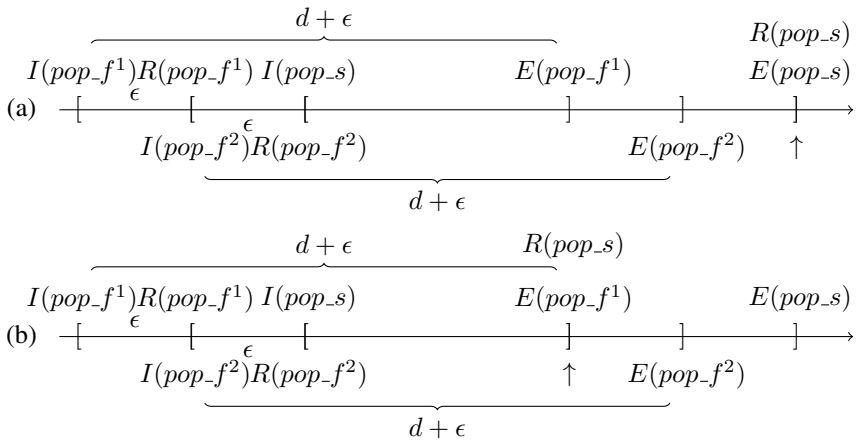

Figure 3.4: (a) Non-short-circuited and (b) short-circuited Pops, with $l - \tau = 2$. In each case, $\uparrow$ indicates the earliest time another Pop instance can be invoked.

This final implementation of the Out-of-Order k-Relaxed Stack is given in Algorithm 2, and is also the version that we will analyze throughout the remainder of this section.

Algorithm 2 Pseudocode for each process p_i implementing an Out-of-Order k-Relaxed Stack, where $k \geq n$.

```
1: HandleEvent PUSH(val)
2:     if localFastPushesActive < τ_c − 1 then
3:         send (push_f, val, ⟨localTime, i⟩) to all
4:         setTimer(ϵ, ⟨push_f, null, ⟨localTime, i⟩⟩, respond)
5:         localFastPushesActive++
6:     else
7:         send (push_s, val, ⟨localTime, i⟩) to all
8:         setTimer(ϵ, ⟨push_s, 1, ⟨localTime, i⟩⟩, respond)
9: HandleEvent POP
```

10: **if** $lStack.peekByLabel(p_i, safeRegion()) \neq \perp$ or $lStack.size() < k - \tau n$ **then**
11: let $ret := lStack.peekByLabel(p_i, safeRegion())$
12: $ret.popped := true$
13: send $(pop\_f, ret, \langle localTime, i\rangle)$ to all
14: $setTimer(\epsilon, \langle pop\_f, ret, null\rangle, respond)$
15: **else**
16: send $(pop\_s, null, \langle localTime, i\rangle)$ to all
17: $setTimer(\epsilon, \langle pop\_s, 1, \langle localTime, i\rangle\rangle, respond)$
18: **HandleEvent** RECEIVE (op, val, ts) FROM p_j
19: $Pending.insert(\langle op, val, ts\rangle)$
20: $setTimer(d + u, \langle op, val, ts\rangle, execute)$
21: **if** $op == pop\_f$ or $op == pop\_s$ **then**
22: $popsInPending{+}{+}$
23: **if** $op == pop\_f$ **then**
24: $resetVolatility(val)$
25: **HandleEvent** EXPIRETIMER($\langle op, val, \langle *, j\rangle\rangle, respond$)
26: **if** $op == pop\_f$ **then return** val
27: **else if** $op == pop\_s$ **then**
28: $flushReservedQueueFront(j)$
29: **if** $reservedQueues[j].peek \neq \perp$ **then**
30: $reservedQueues[j].peek().popped := true$
31: **return** $reservedQueues[j].peek()$
32: **else if** $val \cdot \epsilon - [2d - (l - \tau)\epsilon] < \epsilon$ **then**
33: $setTimer(\min\{\epsilon, 2d - (l - \tau)\epsilon - val \cdot \epsilon\}, \langle pop\_s, val + 1, \langle *, j\rangle\rangle, respond)$
34: **else return** $\perp$
35: **else if** $op == push\_s$ and $localFastPushesActive == \tau_c - 1$ **then**
36: $setTimer(\min\{\epsilon, 2d - (\tau_c - 1)\epsilon - val \cdot \epsilon\}, \langle push\_s, val + 1, \langle *, j\rangle\rangle, respond)$
37: **else return** ACK
38: **HandleEvent** EXPIRETIMER($\langle op, val, ts\rangle, execute$)
39: **while** $ts \geq Pending.min()$ **do**
40: $\langle op', val', ts'\rangle := Pending.extractMin()$
41: $executeLocally(op', val', ts')$
42: $cancelTimer(\langle op', val', ts'\rangle, execute)$
43: **function** EXECUTELOCALLY($op, val, \langle time, j\rangle$)
44: **if** $op == push\_f$ or $op == push\_s$ **then**
45: $lStack.push(val)$
46: **if** $lStack.size() \leq k - \tau n$ **then**
47: $lStack.label(\text{argmin}_{p_h} |lStack.setByLabel(p_h) \cap safeRegion()|, val)$

48: **else**

49: $lStack.label(lStack.getFromTop(k - \tau n + 1).label, val)$

50: $setVolatility(val, lStack.getFromTop(k - \tau n + 1))$

51: $lStack.label(null, lStack.getFromTop(k +$
$\min\{|lowerRegion()|, popsInPending\} + 1)$

52: **if** $j == i$ and $op == push\_f$ **then** $localFastPushesActive--$

53: **else**

54: **if** $j \neq i$ **then** $flushReservedQueueFront(j)$

55: **if** $op == pop\_f$ **then** let $toRemove := val$

56: **else** let $toRemove := reservedQueues[j].dequeue()$

57: **if** $toRemove.higher \neq null$ **then** $resetVolatility(toRemove.higher)$

58: $popsInPending--$

59: **if** $toRemove \in safeRegion()$ **then**

60: **if** $volatileRegion() \cap lStack.topSet(k - \tau n) \neq \emptyset$ **then**

61: let $toRelabel := lStack.peekBySet(volatileRegion())$

62: $resetVolatility(toRelabel)$

63: **else** let $toRelabel := lStack.getFromTop(k - \tau n + 1)$

64: $lStack.label(j, toRelabel)$

65: $reservedQueues[j].enqueue(toRelabel)$

66: $lStack.remove(toRemove)$

67: **if** $lStack.labeledSize() < k$ **then** $lStack.label(dummy, lStack.peekByLabel(null))$

68: **function** SAFEREGION

69: **return** $[topSet(k - \tau n) \cup topSetBySet(popsInPending, lowerRegion())] \setminus$
$topSetBySet(popsInPending, volatileRegion())$

70: **function** VOLATILEREGION

71: **return** $\{elem \in lStack.topSet(k - \tau n) : elem.t_{volatile} > localTime - (2d + 2u + \epsilon)\}$

72: **function** LOWERREGION

73: **return** $\{elem.lower : elem \in volatileRegion()\}$

74: **function** SETVOLATILITY($elem, other$)

75: $elem.t_{volatile} := localTime$

76: $elem.lower := other$

77: $elem.lower.higher := elem$

78: **function** RESETVOLATILITY($elem$)

79: $elem.t_{volatile} := -\infty$

80: $elem.lower.higher := null$

81: $elem.lower := null$

82: **function** FLUSHRESERVEDQUEUEFRONT(j)

83: let $front := reservedQueues[j].peek()$
84: **while** $\neg lStack.contains(front)$ or $front.popped$ or $front.label \neq p_j$ **do**
85: $reservedQueues[j].dequeue()$
86: let $front := reservedQueues[j].peek()$

3.1.3 Correctness

Let $ts(op)$ denote the timestamp associated with an operation instance op given as the first argument in line 3, 7, 13, or 16.

Construction 1. *Define the permutation π of operation instances in a complete, admissible run of Algorithm 2 as the order given by sorting by $ts(op)$ for each operation instance op.*

We define the *pending delay*, which is the amount of time that passes between the receipt of a message sent by the invocation of an operation instance and the local execution of that operation instance. In Algorithm 2, the pending delay is $d + u$ (line 19).

The following lemmas show that Construction 1 respects the real-time partial order of operation instances, following a similar method as in [11].

Lemma 1. *If the pending delay of an algorithm is at least $u + \epsilon$, then each process running the algorithm locally executes all $Pushes$ and $Pops$ in timestamp order.*

Proof. Let p be the pending delay, which is at least $u + \epsilon$.

Each operation instance op_1 upon invocation sends a message to all processes, each of which receives this message within $d - u$ to d time. Each receiving process, including the sender, then inserts the operation instance into its local $Pending$ priority queue and sets a timer to the pending delay p. The process executes op_1 if the timer expires, so each operation instance is locally executed at each process no later than $d + p = d + u + \epsilon$ real time after it is invoked.

op_1 may be executed earlier than the expiration time of its timer, however, if the $execute$ timer for another instance, op_2, with $ts(op_2) > ts(op_1)$ expires sooner. But the earliest time

the timer may expire is $(d-u)+p = d+\epsilon$ real time after op_2 is invoked, and since the maximum message delay is d and op_1 could have been invoked no later than ϵ real time after op_2, the sum of the delays is $d+\epsilon$, the latest in real time after op_1's invocation that op_1 could have been inserted into $Pending$. Since $ts(op_1) < ts(op_2)$ by assumption, op_1 is in $Pending$ before the timer for op_2 expires, so it is executed before op_2. □

Lemma 2. *Each process running Algorithm 2 locally executes all $Push$es and Pops in timestamp order.*

Proof. Since $d+u > u+\epsilon$, Lemma 1 holds for Algorithm 2. □

Lemma 3. *π respects the real-time order of non-overlapping operation instances.*

Proof. If we have two instances op_1 and op_2 with $ts(op_1) < ts(op_2)$, then op_1 responds before op_2 does, because the time that passes between invocation and response is always at least ϵ for all operation instances in Algorithm 2, while the difference between the local clocks of the invoking processes is at most ϵ. □

We proceed to introduce a number of new definitions and establish essential properties of Algorithm 2 which will appear repeatedly and remain useful throughout this paper.

Lemma 4. *There are at most τ_c mutually active $Push$es occurring in a single process and at most $\tau_c n$ mutually active $Push$es occurring over all processes at any given time.*

Proof. In a process p_i, since each fast $Push$ increments $localFastPushesActive$ at invocation (line 5) and decrements it at local execution (line 5), $localFastPushesActive$ counts the number of fast $Push$es currently taking place in process p_i. Because the number of mutually active fast $Push$es at p_i is limited to $\tau_c - 1$ (line 2), a τ_cth mutually active $Push$ must be slow, and we show in the proof of Theorem 15 in Section 3.1.4 that between the invocation and response of a slow $Push$, at least one fast $Push$ has executed locally, so the invocation of the next $Push$ increases the number of mutually active $Push$es to at most τ_c. Since at most τ_c $Push$es in total may be mutually active in p_i, we conclude that among all processes, there can be up to $\tau_c n$ mutually active $Push$es. □

Definition 11. *Elements are* safe *if they are legal to return by $Pops$ that are at any point between invocation and local execution, given that they are not already returned.*

Lemma 5. *The $safeRegion()$ function given in Algorithm 2 returns a set of elements that are guaranteed to be safe. We also refer to this set as the* safe region*.*

Proof. We can deduce from Lemma 4 that τn is an upper bound for the number of active $Push$es in each process, while $popsInPending$ is a lower bound for the number of active Pops since for each Pop, it is incremented once after invocation (line 22) and decremented once at local execution (line 58). Then $k - \tau n + popsInPending$ is a lower bound for the number of topmost elements that are legal to return. We can see that $safeRegion()$ is contained in this set of elements. □

Definition 12. *The* uniformity invariant *states that*

$$|lStack.setByLabel(p_i) \cap safeRegion()| \leq \left\lceil \frac{k}{n} \right\rceil - \tau \quad \forall i \in [0, n-1]$$

with the number of i such that the bound is reached being at most $n_a := k - (\lceil k/n \rceil - 1)n$, and when $lStack.size() \geq k - \tau n$,

$$|lStack.setByLabel(p_i) \cap safeRegion()| - |lStack.setByLabel(p_j) \cap safeRegion()| \leq 1$$

$\forall i, j \in [0, n-1]$.

Definition 13. *Any element in the local stack is* volatile *if it is contained in $volatileRegion()$; in other words, its $t_{volatile}$ field was set less than $2d + 2u + \epsilon$ time before the current local time (line 91).*

Definition 14. *If the size of the local stack is at least k and a $Push$ executes locally, then as long as the new top element $higher$ remains volatile, $higher$ and the new $(k-\tau n+1)$th-topmost element $lower$ form a $(higher, lower)$-*pair.

The volatile region *is the set of volatile elements, or alternately the set of all elements that are the* $higher$ *element in some* $(higher, lower)$*-pair, in the topmost* $k - \tau n$ *elements. The* lower region *is the set of all elements that are the* $lower$ *element in some* $(higher, lower)$*-pair where* $higher$ *is in the volatile region.*

Lemma 6. *If an element* $elem$ *is the* $lower$ *element in some* $(higher, lower)$*-pair, then the value of* $elem.higher$ *is the other element; if not,* $elem.higher$ *is* $null$*. Likewise, if* $elem$ *is the* $higher$ *element in some* $(higher, lower)$*-pair, then the value of* $elem.lower$ *is the other element, and if not,* $elem.lower$ *is* $null$*.*

Proof. $(higher, lower)$-pairs are created only by locally executing $Push$es that increase the size of the local stack to greater than $k - \tau n$ (line 50 is the only place where elements are made volatile). The $lower$ element in the new $(higher, lower)$-pair is some element that already exists in the local stack. Then lines 76 and 77 establish that $higher.lower = lower$ and $lower.higher = higher$.

Events that cause a $(higher, lower)$-pair to cease to exist by definition include the removal of either element or the loss of $higher$'s volatility. By definition, the $remove()$ operation on the local stack sets $higher.lower$ and $lower.higher$ to $null$. Furthermore, we can see from line 78 that a generic function call $resetVolatility(higher)$ sets $higher.lower$ and $lower.higher$ to $null$ and makes $higher$ non-volatile at once.

Lastly, there will not be a volatile element $elem$ with $elem.lower = null$, because the only time the $lower$ element of a $(higher, lower)$-pair can be removed is at the local execution of a Pop, at which $higher$ is made non-volatile (line 57). □

Definition 15. *A set of elements in the local stack is* connected *or are* contiguous *if as long as it contains no element in the topmost* m *elements but not the topmost* $m-1$ *elements for any* $m > 1$*, then it contains no element in the topmost* $m + 1$ *elements but not the topmost* m *elements.*

Lemma 7. *The* volatility invariant *holds, and states that in the local stack of each process,*

(i) any two elements that form a $(higher, lower)$*-pair share the same label;*

(ii) if an element $elem$ *is the topmost in the volatile region, then* $elem.lower$ *is currently the* $(k - \tau n + 1)$*st-topmost element, so that any element is the* m*th-topmost volatile element if and only if it is the* $higher$ *of a* $(higher, lower)$*-pair in which* $lower$ *is the* m*th-topmost element in* $lowerRegion()$*;*

(iii) the lower region is mutually exclusive with the topmost $k - \tau n$ *elements and connected; and*

(iv) member elements of a $(higher, lower)$*-pair cannot mutually exist in the safe region.*

Proof. The statements rely on the fact that $(higher, lower)$-pairs are created only by locally executing $Push$es.

(i) Whenever a $(higher, lower)$-pair is created, $higher$ is given the same label as $lower$ (line 49). Whenever any element is relabeled, if it is the $higher$ element of a $(higher, lower)$-pair, then it is made non-volatile (line 62), and if it the $lower$ element of a $(higher, lower)$-pair, then its $higher$ element is made non-volatile (line 57); the $(higher, lower)$-pair necessarily no longer exists in either case.

(ii) In a series of $Push$es, each newly added volatile element must be the new top element and hence the topmost volatile element in the local stack, and each new $(k-\tau n+1)$-st topmost element which is added to the lower region is also higher up than the previous $(k - \tau n + 1)$-st element in the local stack. Any Pop that makes a volatile element non-volatile must do so for the topmost volatile element, and also moves the previously $(k - \tau n + 1)$st-topmost element to the $(k - \tau n)$th place, so that the new $(k - \tau n + 1)$st-topmost element is now the topmost member of the lower region if $(higher, lower)$-pairs still exist, thereby preserving the stated property.

(iii) An element is made the $lower$ element of a $(higher, lower)$-pair if and only if it becomes the $(k-\tau n+1)$st-topmost element due to a $Push$ (line 76), and subsequent $Push$es only move existing lower region elements farther from the topmost $k - \tau n$

while maintaining their adjacency since they are all moved down simultaneously in the local stack.

Only a Pop may bring a lower region element $elem$, which must be the topmost, back into the topmost $k - \tau n$ elements, but such a Pop would also make the topmost volatile element non-volatile (line 62), and by (ii) it is $elem.higher$, which implies that $elem$ is no longer the $lower$ element in a $(higher, lower)$-pair. While $elem$ is no longer part of the lower region, the remaining elements in the lower region are still outside the topmost $k - \tau n$ and contiguous.

(iv) This follows from (ii) and the definition of $safeRegion()$ (line 69).

□

Corollary 8. *The size of the safe region is always* $\min\{k - \tau n, lStack.size()\}$.

Proof. From Lemma 6, we know that every volatile element $elem$ has $elem.lower \neq null$, so $volatileRegion()$ and $lowerRegion()$ are of the same size. In the definition of $safeRegion()$ (line 69), since the top $k - \tau n$ elements are mutually exclusive with the lower region by (iii) from Lemma 7, the size of their union is the sum of their separate sizes. Taking the union's set difference with a set contained in the union of the same size as the lower region results in a set of size $k - \tau n$. □

Corollary 9. *Incrementing or decrementing* $popsInPending$ *preserves the uniformity invariant.*

Proof. First, we consider incrementing. If there are no volatile elements in the safe region, then the safe region by its definition is left unchanged when $popsInPending$ is incremented. Otherwise, if there are volatile elements in the safe region, meaning they are not the topmost $popsInPending$ volatile elements, then incrementing $popsInPending$ moves an element of the volatile region out of the safe region and moves another element of the lower region into the safe region. From Lemma 7, (iv) requires that these elements form a

$(higher, lower)$-pair, and thus they share the same label by (i). The number of elements with each label is preserved.

Next, we consider decrementing. If all the volatile elements are in the safe region, then the safe region by its definition is left unchanged when $popsInPending$ is decremented. Otherwise, if there are volatile elements outside the safe region, meaning they are among the topmost $popsInPending$ volatile elements, then decrementing $popsInPending$ moves an element of the volatile region into the safe region and moves another element of the volatile region out of the safe region. Then by the same reasoning as before, the uniformity invariant holds. □

Remark 1. $popsInPending$ is incremented by 1 (line 22) every time the process receives a message sent from any Pop and is decremented by 1 (line 58) every time any Pop executes locally, which means this variable counts the number of active Pops from which this process received a message.

Definition 16. *The* labeling invariant *states that the labeled elements in the local stack are the topmost* $k + \min\{|lowerRegion()|, popsInPending\}$ *elements.*

The arguments required in the next lemma form the most subtle part of this proof of correctness and inspire the more complicated procedures of the algorithm in large part, making use of message passing.

Lemma 10. *Any element returned by a* Pop *will not be relabeled with another process id.*

Proof. We consider the process p_i. Only Pops relabel elements with new process ids, and a Pop can only relabel an element $elem$ if it is either volatile in the topmost $k - \tau n$ elements (line 61) or, if there is no such element, the new $(k - \tau n)$th-topmost element in p_i's local stack (line 63).

In the former case, the volatile element $elem$ can only be returned by a Pop, which we call pop, invoked by p_i at time t_0 if pop is fast and the element is in the safe region, which from its definition means that it is not in the topmost $popsInPending$ elements. Since

each Pop contributing to $popsInPending$ relabels only the top volatile element (line 61) at the time of its local execution, it will not relabel $elem$. Also, receiving a message takes at most d time, so any active Pop from which p_i has not received a message by time t_0, and hence does not contribute to $popsInPending$, will not locally execute until after $t_0 + (2d - d) = t_0 + d$. Lastly, *pop* sends a message at time t_0 to all other processes to make $elem$ non-volatile (line 24), which they receive by time $t_0 + d$. Thus, none of the active Pops have the risk of relabeling $elem$.

In the latter case where *pop*'s invocation claims an element $elem$ to return which becomes the $(k - \tau n)$th-topmost element in p_i's local stack resulting from another locally executing Pop we call pop_2, then pop_2 cannot have relabeled $elem$ unless there were no volatile elements in the topmost $k - \tau n$ (line 60). *pop* must have claimed $elem$ as an element in the safe region. By the assumption of pop_2's existence, at least one of the following must be true: (i) $elem$ was outside the topmost $k - \tau n$ elements at the time of *pop*'s invocation, which means that the topmost $k - \tau n$ elements had volatile elements and $popsInPending > 0$ by the definition of $safeRegion()$, or (ii) some locally executing $Push$ moved $elem$ outside the topmost $k - \tau n$ elements in p_i's local stack between *pop*'s invocation and pop_2's local execution in p_i.

Given that (i) is true, all the Pops contributing to $popsInPending$ must execute locally within $d + u$ time from *pop*'s invocation since they were invoked at least d real time ago, and the local executions remove a number equal to $popsInPending$ of elements which moves $elem$ into the topmost $k - \tau n$ elements, unless another $Push$ is locally executed, the case in which is handled in (ii).

Consider when (ii) is true. We know that at each process, *pop* may locally execute up to $2d + u$ real time after invocation, because it can take up to d time for *pop*'s message to arrive at a process, and the $execute$ timer is set to $d + u$. The local execution of the $Push$, which succeeds *pop*'s invocation in p_i but which may happen $u + \epsilon$ time earlier in real time in other processes, makes the newly added element volatile for $2d + 2u + \epsilon$ time in each

process, so the element is still volatile in each process by the time pop executes locally. Then this element is relabeled instead of $elem$. □

The $reservedQueues[]$ local variable helps guarantee that the element returned by p_i for the slow Pop will be the same element that is removed in every process' local stack upon the slow Pop's local execution, so that every process maintains the same local state. We state this in the following lemma.

Lemma 11. *If slow $Pops$ invoked by p_i respond before local execution, then any element returned by a slow Pop will be the same element that is removed from every process' local stack during local execution.*

Proof. Let *pop* denote a slow Pop. If we show that calling $flushReservedQueueFront(i)$ in p_i by *pop*'s response (line 28) and in all other processes by *pop*'s local execution (line 54) removes the same elements from $reservedQueues[i]$ both in p_i and in every other process, then we guarantee that $reservedQueues[i].peek()$ at the response (line 31) and $reservedQueues[i].dequeue()$ at the local execution (line 56) give the same element. This is done casewise by considering each predicate for removing elements from $reservedQueues[i]$ in the definition of FLUSHRESERVEDQUEUEFRONT (line 84):

- $\neg lStack.contains(front)$: If an element is removed from the front of $reservedQueues[i]$ in p_i by *pop*'s response due to being already removed from p_i's local stack, then it is also removed from $reservedQueues[i]$ in every process by *pop*'s local execution, because it is also already removed from every process' local stack, as elements are only removed during EXECUTELOCALLY in all processes.

  We show the converse that if an element is removed from $reservedQueues[i]$ in every process by *pop*'s local execution due to being already removed from every process' local stack, then it has also been removed from $reservedQueues[i]$ in p_i by *pop*'s response. This is done by induction, starting with the base case that if *pop* is mutually active with no other slow $Pops$, then an element already removed from

every process' local stack before pop's local execution must have been removed by an earlier-invoked fast Pop. The fast Pop must have returned the same element and marked it $popped$ before pop's response, fulfilling the condition to remove it from $reservedQueues[i]$ in p_i by pop's response, since the response time of fast Pops is ϵ, which is at most that of slow Pops.

The inductive step is that if we assume the converse statement holds for a certain number of mutually active slow Pops, then it holds if pop is an additional mutually active slow Pop invoked later than the rest. Slow Pops invoked by other processes do not use the ith index of $reservedQueue[]$, so if there are no slow Pops invoked by p_i then the reasoning is the same as the base case. Assume that there are mutually active slow Pops invoked by p_i, and let pop_{prev} denote the previous such slow Pop. If an element $elem$ is removed from $reservedQueues[i]$ in all processes due to being already removed in their local stacks sometime between the local executions of pop_{prev} and pop, then $elem$ was not removed from $reservedQueues[i]$ in p_i at the time of pop_{prev}'s response by the first proposition of this bullet point. $elem$ must, however, be removed from $reservedQueues[i]$ in p_i by pop's response, either because of an earlier-invoked fast Pop (the reasoning proceeds like the base case) or because it was returned and marked $popped$ (line 30) by pop_{prev}, since these are the only types of operation instances that can remove $elem$ from every process' local stack before pop executes locally. Assuming the latter case, since pop_{prev} must respond before pop is invoked and hence before pop responds, $elem$ is removed from $reservedQueues[i]$ in p_i by pop's response.

- $front.popped$: If an element $elem$ labeled p_i is $popped$, then it must have returned by a Pop, and $elem.popped = true$ only in process p_i (line 12). This means that if an element is removed from $reservedQueues[i]$ in p_i by pop's response due to being $popped$ in p_i, then it must have been returned by a Pop invoked in p_i before pop was invoked, and hence will execute locally before the slow Pop does because all

operation instances invoked by p_i execute locally in p_i in the same amount of time, $2d$. Hence, the element will be removed from the local stack in all processes on line 54.

Considering the converse, $reservedQueues[i]$ never contains an $popped$ element in any process other than p_i, because if it is labeled p_i, it is marked $popped$ only in p_i.

- $front.label \neq p_j$: By Lemma 10, an element already returned by a Pop is never relabeled. Then between pop's response in p_i and local execution in each process, whether an element in $reservedQueues[i]$ is still labeled p_i remains constant.

We have shown that if an element is removed from $reservedQueues[i]$ in p_i at pop's response, then it is removed from $reservedQueues[i]$ in all other processes at pop's local execution, and vice versa. Therefore, the element p_i returns on line 31 is the same element that all processes share at the front of their reserved queues on line 56, which they will then remove from their local stacks. □

By inductively showing that the local execution of each operation type maintains a legal linearization and satisfies the invariants defined thus far, we demonstrate that any execution of Algorithm 2 when linearized produces a legal sequence of operation instances according to Definition 2.

Theorem 12. *For any execution of Algorithm 2, the permutation π given by Definition 1 is legal by the specification of an Out-of-Order k-Relaxed Stack. Thus, Algorithm 2 is a correct implementation of an Out-of-Order k-Relaxed Stack.* □

Proof. We consider local execution by operation type, and show that for every sequence of operation instances invoked by the processes, the algorithm (a) generates return values such that π is legal, and also maintains (b) the labeling invariant and (c) the uniformity invariant. (b) and (c) are demonstrated by induction on the local stack, with the base case being that the invariants are vacuously satisfied when the stack is empty.

$Push$:

(a) As long as every value can be uniquely identified (such as in the method suggested in the footnote under Definition 1), *(C2)* in Definition 1 is satisfied and hence any $Push$ is legal.

(b) We assume inductively that the elements of the current local stack for any process exhibit a labeling satisfying the labeling invariant before the $Push$ executes locally.

If the size of the local stack is at most k following the $Push$, then the $Push$ed element is labeled with a new label (line 47 – chosen to be the process id such that the fewest elements are labeled with it) in order to preserve the labeling invariant, which states that all the elements in the local stack when it is of this size must be labeled because they are legal for a Pop to return.

Otherwise, the new topmost element is labeled with the label belonging to the element that just left the safe region (line 49), and it is justified to unlabel the element given on line 51 by the following argument. By (iii) in the volatility invariant, the bottommost element $bottom$ in the safe region is the $(k - \tau n + \min\{|lowerRegion()|, popsInPending\})$th-topmost, and any action that decreases either operand in the $\min$ function must be a Pop that decreases it by 1 so that $bottom$ moves closer to the top by 1, so by Lemma 4 no mutually active $Push$ will move a safe region element returned by a Pop out of the topmost k elements in the local stack. Therefore, the top k and only the top k elements are labeled, and the the labeling invariant is maintained.

(c) Assume that the uniformity invariant holds before the $Push$ locally executes.

If the local stack still has at most $k - \tau n$ elements after the element is added by the locally executed $Push$ (line 46), then we first state that every element is in the safe region – volatile elements exist only if the size of the local stack is greater than $k - \tau n$, since only in this case do locally executing $Push$es make them volatile; also, locally executing Pops that reduce the size of the stack make volatile elements

non-volatile. Without volatile elements, the safe region is simply the set of $k - \tau n$ topmost elements.

Then, by the pigeonhole principle, fewer than $\lceil (k - \tau n)/n \rceil = \lceil k/n \rceil - \tau$ elements have the label shared by the smallest number of elements (chosen on line 47) before the $Push$. This increases to at most or fewer than $\lceil k/n \rceil - \tau$ following the $Push$, depending on whether the number of processes such that $\lceil k/n \rceil - \tau$ elements are labeled with its id is fewer than n_a before the $Push$.

Also, if after the $Push$ the local stack has exactly $k - \tau n$ elements, then they are all labeled. If $n \mid k$ then every process must have $\lceil k/n \rceil - \tau = k/n - \tau$ elements labeled with its id, or the pigeonhole principle would require that some process has more than $\lceil k/n \rceil - \tau$ elements, violating the uniformity invariant. We can see that in this case, the second part of the uniformity invariant is maintained, that the number of elements labeled with each label differ by at most one.

If $n \nmid k$ and n'_a denotes the number whose label is shared by $\lceil k/n \rceil - \tau$ elements, then the uniformity invariant implies that the number of elements with labels shared by fewer than $\lceil k/n \rceil - \tau$ elements is at least

$$
\begin{aligned}
(k - \tau n) - (\lceil k/n \rceil - \tau) n_a &= \left(\frac{k - \lceil k/n \rceil n_a}{n - n_a} - \tau \right) (n - n_a) \\
&= \left(\frac{k - \lceil k/n \rceil [k - (\lceil k/n \rceil - 1) n]}{\lceil k/n \rceil n - k} - \tau \right) (n - n_a) \\
&= \left(\frac{k - (\lfloor k/n \rfloor + 1)(k - \lfloor k/n \rfloor n)}{(\lfloor k/n \rfloor + 1) n - k} - \tau \right) (n - n_a) \\
&= \left(\frac{k - (\lfloor k/n \rfloor + 1)(k - \lfloor k/n \rfloor n)}{(\lfloor k/n \rfloor + 1) n - k} - \tau \right) (n - n_a) \\
&= \left(\frac{-k \lfloor k/n \rfloor + \lfloor k/n \rfloor^2 n + \lfloor k/n \rfloor n}{(\lfloor k/n \rfloor + 1) n - k} - \tau \right) (n - n_a) \\
&= (\lfloor k/n \rfloor - \tau) (n - n_a),
\end{aligned}
$$

and we know that the number of processes whose id is given by one of these labels

is $n - n'_a$. In fact, the above bound is tight and $n'_a = n_a$ because otherwise, by the pigeonhole principle, one of these processes would have at least $\lceil k/n \rceil$ elements, contradicting its definition. Using the fact that $|\lceil k/n \rceil - \lfloor k/n \rfloor| = 1$ shows that the second part of the uniformity invariant is also maintained in this case.

In the remaining case that the local stack has greater than $k - \tau n$ elements, the element new to the safe region is the $higher$ element in a $(higher, lower)$-pair, so by the volatility invariant, the $lower$ element shares the same label and is the only element that leaves the safe region. The number of elements in the safe region with each label is preserved, so the uniformity invariant holds.

Pop:

(a) We now show that each Pop instance satisfies *(C3)* in Definition 2.

First, a Pop invoked by a process p_i only returns an element labeled with the process' id, and subsequently makes the element $popped$ so that p_i will not again return the same element in a later Pop. Because $Push$es add unique elements and Lemma 11 states that a Pop that returns an element in the local stack removes the same element from each local stack, Pops return unique values.

Since every process executes operation instances in linearized order, the second part of *(C3)* is equivalent to stating that the element removed by the Pop is among the topmost k in every process' local stack at the time of the Pop's local execution.

During invocation in a process p_i, a Pop instance, which we denote pop, finds (line 10) whether the safe region of p_i's local stack contains a non-$popped$ element val labeled p_i. If so, val is legal to return and pop is a fast Pop, because by (iii) in Lemma 7, the bottommost element in the safe region of the local stack when a fast Pop occurs is the $(k-\tau n+popsIncoming)$th-topmost in the local stack; therefore, it will not be moved outside the topmost k elements by any of the at most $\tau_c n$ mutually active $Push$es that precede pop in π (if they exist) without being moved back in by

any of the at least $popsIncoming$ mutually active Pops that precede pop in π (if they exist). If not, but the safe region constitutes the entire local stack (line 10), then $\perp$ will be returned, which is legal by the definition of an Out-of-Order k-Relaxed Stack because the mutually active $Push$es, being limited to $\tau_c n$ in number, will not cause the size of the local stack to become k or greater.

In the remaining case, where val does not exist and there are elements outside the safe region, pop is a slow Pop. A slow Pop is only invoked in p_i if the size of the local stack is at least $k - \tau n$ and all the elements in the safe region labeled p_i are $popped$ (line 10), which means at least that from the uniformity invariant, at least $l - \tau$ fast Pops (line 12) have responded but not yet executed locally.

Any of the fast Pops at local execution relabels either a volatile element within the topmost $k-\tau n$ elements in the local stack if one exists (line 61) or the new $(k-\tau n)$th element (line 63) p_i, adding it to the safe region by its definition. It also enqueues the element in the reserved queue for process p_i (line 65), which pop may then return (line 56).

pop's response occurs within ϵ time of the element being added to the reserved queue since the timer set at invocation is reset at ϵ-length time intervals before the maximum waiting time has passed (line 33). No more than τn $Push$es could have executed locally by Lemma 4 and the observation that each process cannot locally execute more than one operation instance in an ϵ-length time interval. Therefore, the returned element is still within the topmost k in the local stack at the time of pop's response and is hence legal to return.

If $\perp$ is returned after the maximum time $2d - (l - \tau)\epsilon$ has passed between pop's invocation and response, which is the latest time that the first fast Pop may return since it takes $(l - \tau)\epsilon$ time for $l - \tau$ fast Pops to invoke and return in the same process and each executes locally in $2d$ time, then we may deduce that the fast Pops

each found no volatile element to relabel and the $(k - \tau n)$th element was $\perp$; thus, the local stack was of size less than $k - \tau n$, and $\perp$ is legal to return.

(b) By induction, the labeling done by Pops satisfies the labeling invariant, because if after the local execution of a Pop removes an element from the local stack it subsequently has fewer than k labeled elements, then it is necessary and sufficient to label the topmost unlabeled element if one exists (lines 67). If the local stack has at least k labeled elements on the other hand, then by the definition of the labeling invariant there must previously have been volatile elements. Since the Pop decrements $|lowerRegion()|$ by 1 by removing a volatile element and also decrements $popsInPending$ by 1, the labeling invariant is automatically maintained in this case.

(c) If the local stack had at most $k - \tau n$ elements before the Pop executed locally, then the number of elements with some label is decreased following the Pop and none are increased, so the first part of the uniformity invariant still holds, and the second part does not apply. We are also done if the element removed by the Pop was in the contention region at the time of Pop's local execution, because in this case decrementing $popsInPending$ (line 58) is the only change made to the safe region, and it preserves the uniformity invariant by Corollary 9.

Assuming that the none of the above conditions hold, the local execution of the Pop moves an element from the contention region to the safe region; since the number of elements labeled p_j has decreased by one, labeling some element p_j which becomes part of the safe region allows the uniformity invariant to again hold. Depending on whether there are any volatile elements, this element is chosen to be either the newly-entered element in the safe region from the bottom (line 63) or some element in the volatile region sharing the same label (line 61) which is then made non-volatile. In the latter case, if the element was outside the safe region, then this action together with decrementing $popsInPending$ (line 58) brings this element into the safe region.

Lastly, we show that message receipts from Pops preserve the uniformity invariant. The execution that increments $popsInPending$ (line 22) upon receiving such a message satisfies the invariant by Corollary 9, and the execution upon receiving a message from a fast Pop (line 24) only makes an element that was already in the safe region (line 24) non-volatile, so it does not modify the set returned by $safeRegion()$. □

3.1.4 Performance

Our discussion of performance clarifies the purpose of defining the effective quantities given at the beginning of this section.

We first consider the $Push$ operation. The first result immediately follows from observing the response time for fast $Push$es (line 4) and the two different possible response times for slow $Push$es (lines 8 and 36, with the second being the sum) in Algorithm 2.

Theorem 13. *The worst-case time complexity of $Push$ among all complete, admissible executions of Algorithm 2 is no more than* $\max\{2d + (1 - \tau_c)\epsilon, \epsilon\}$. □

Remark 2. At least $\tau_c - 1$ fast $Push$es, which together take at least $(\tau_c - 1)\epsilon$ time (if each is executed one right after another with no gaps in between), must be invoked by a single process and not yet executed locally before the next $Push$ is slow. We note that this is impossible if $2d < (\tau_c - 1)\epsilon$ implying $d < (\tau_c - 1)\epsilon/2$; in this case, the first operand in the upper bound given in Theorem 13 would become negative, which would have been an invalid result. We reach the interesting conclusion that all $Push$es are fast when and only when $d < (\tau_c - 1)\epsilon/2$.

To lead up to our next theorem, we consider the view of a single process p_i in which occurs a slow $Push$, i.e., the τ_cth in a subsequence σ of $Push$es contained in the sequence of operation instances invoked by p_i in which less than $2d$ time has passed since the invocation of the first instance. We also require σ to consist of at least $2\tau_c$ $Push$es.

Let t_0 denote the time of invocation of the first instance in σ, and let gap_x denote the amount of time that passes between the time the xth instance returns and the time the $(x+1)$st instance is invoked for $x \in [1, \tau_c - 1]$. Then the time of invocation of the slow $Push$ is $t_0 + (\tau_c - 1)\epsilon + \sum_{m=1}^{\tau_c - 1} gap_m$. From Theorem 13, the slow $Push$ returns in $\max\{2d + (1-\tau_c)\epsilon, \epsilon\}$ time, so the time between its invocation and return is

$$\begin{aligned} & t_0 + (\tau_c - 1)\epsilon + \sum_{m=1}^{\tau_c - 1} gap_m + \max\{2d + (1-\tau_c)\epsilon, \epsilon\} \\ &= t_0 + \sum_{m=1}^{\tau_c - 1} gap_m + \max\{2d, \tau_c\epsilon\}. \end{aligned} \tag{3.1}$$

In the following argument, we look ahead to the $\tau_c - 1$ $Push$ instances that follow after the τ_cth in σ.

Lemma 14. *For $2 \leq x \leq \tau_c$, the latest time the xth $Push$ executes locally is ϵ time after the $(\tau_c + x - 1)$st $Push$ is invoked.*

Proof. First, consider the invocation of the $(\tau_c + 1)$st $Push$ instance in σ. Since the earliest time it can be invoked is given by (3.1), and the second instance in σ is invoked at time $t_0 + \epsilon + gap_1$ and executed locally at time

$$t_0 + \epsilon + gap_1 + d + \epsilon,$$

the difference in times between the local execution of the second instance and the invocation of the $(\tau_c + 1)$st instance is $\epsilon - \sum_{m=2}^{\tau_c - 1} gap_m \leq \epsilon$.

Continuing the argument by induction for $2 \leq x \leq \tau_c - 1$, if the latest time the xth instance executes locally is ϵ time after the $(\tau_c + x - 1)$st instance is invoked, then, since the $(x+1)$st instance executes locally at time $gap_x + \epsilon$ after the xth instance does so while the earliest time the $(\tau_c + x)$th instance in σ can be invoked is ϵ after the $(\tau_c + x - 1)$st is invoked, we have that the difference in times between the local execution of the $(x+1)$st instance and the invocation of the $(\tau_c + x)$th instance is less than $\epsilon - \sum_{m=x+1}^{\tau_c - 1} gap_m \leq \epsilon$.

□

Theorem 15. *The amortized time complexity of $Push$ among all complete, admissible executions of Algorithm 2 is no more than* $\max\{2d/\tau_c, \epsilon\}$. □

Proof. From Remark 2, if $d < (\tau_c - 1)\epsilon/2$, then every $Push$ takes ϵ time, giving an amortized time complexity of ϵ (which satisfies the proposition since $2d/\tau_c < \epsilon$), so we assume that $d \geq (\tau_c - 1)\epsilon/2$ so that it is possible for slow $Push$es to occur.

With Lemma 14 in hand, we now know that the $\tau_c - 1$ instances following the τ_cth instance also execute locally in ϵ time, the fastest possible. This is because for $2 \leq x \leq \tau_c$, if the $(x + 1)$st instance in σ is executed locally before the $(\tau_c + x)$th is invoked, then the $(x + 1)$st instance decrements $localFastPushesActive$ (line 52), guaranteeing that the $(\tau_c + x)$th instance is a fast $Push$. If the $(x + 1)$st instance is locally executed after the latter is invoked, then it must be within ϵ time, and thus in all processes the former decrements $localFastPushesActive$ soon enough that the latter is able to respond ϵ time after invocation, as equally as fast as a fast $Push$.

The first $Push$ instance after the τ_cth that may require a response time of more than ϵ is the $2\tau_c$th, and so by Theorem 13 it returns in at most $\max\{2d + (1 - \tau_c)\epsilon, \epsilon\}$ time. Thus, we have a repeating pattern of τ_c instances, and the amortized cost of each repetition is

$$\frac{(\tau_c - 1)\epsilon + \max\{2d + (1 - \tau_c)\epsilon, \epsilon\}}{(\tau_c - 1) + 1} = \frac{\max\{2d, \tau_c\epsilon\}}{\tau_c} = \max\left\{\frac{2d}{\tau_c}, \epsilon\right\}.$$

The cost of any prefix of the infinite repetition of this pattern is bounded above by the maximum average cost of a single copy of this pattern, since prefixes ending with a slow $Push$ which ends a pattern have the highest average cost.

Because this provides an upper bound on the cost of $Push$es at any process p_i, it is also an upper bound on the average cost of $Push$es at all processes; hence, it is the amortized cost. □

The Pop operation mirrors the $Push$ operation in that a slow Pop in any sequence of Pop instances follows a series of fast Pops, but the minimum number of fast operation instances is now $l - \tau$ instead of $\tau_c - 1$. The first result comes directly from simplifying the response time for fast Pops (line 14) and the two different possible response times for slow Pops (lines 17 and 33) in Algorithm 2.

Theorem 16. *The worst-case time complexity of Pop among all complete, admissible executions of Algorithm 2 is no more than* $\max\{2d + (\tau - l)\epsilon, \epsilon\}$. □

Remark 3. At least $l - \tau$ fast Pops, which together take at least $(l - \tau)\epsilon$ time (if each is executed one right after another with no gaps in between), must be invoked by a single process and not yet executed locally before the next Pop is slow. We note that this is impossible if $2d < (l - \tau)\epsilon$ implying $d < (l - \tau)\epsilon/2$; in this case, the first operand in the upper bound given in Theorem 16 would become negative, which would have been an invalid result. We reach the interesting conclusion that all Pops are fast when and only when $d < (l - \tau)\epsilon/2$.

Lemma 17. *For $2 \leq x \leq l - \tau + 1$, the latest time the xth Pop executes locally is ϵ time after the $(l - \tau)$th Pop is invoked.*

Theorem 18. *The amortized time complexity of Pop among all complete, admissible executions of Algorithm 2 is no more than* $\max\{2d/(l - \tau + 1), \epsilon\}$. □

Proof. The reasoning is the same as for the amortized time complexity of $Push$ (Theorem 15), and the associated calculation is as follows:

$$\frac{(l-\tau)\epsilon + \max\{2d + (\tau - l)\epsilon, \epsilon\}}{(l-\tau)+1} = \frac{\max\{2d, (l-\tau+1)\epsilon\}}{l-\tau+1} = \max\left\{\frac{2d}{l-\tau+1}, \epsilon\right\}.$$

□

We can now remark that during usage of the Out-of-Order k-Relaxed Stack it is always worth tuning the τ parameter to strike a fine balance between the performance of $Push$

and Pop, since their amortized time complexities are both concave functions. This fact is further illustrated in the plot below.

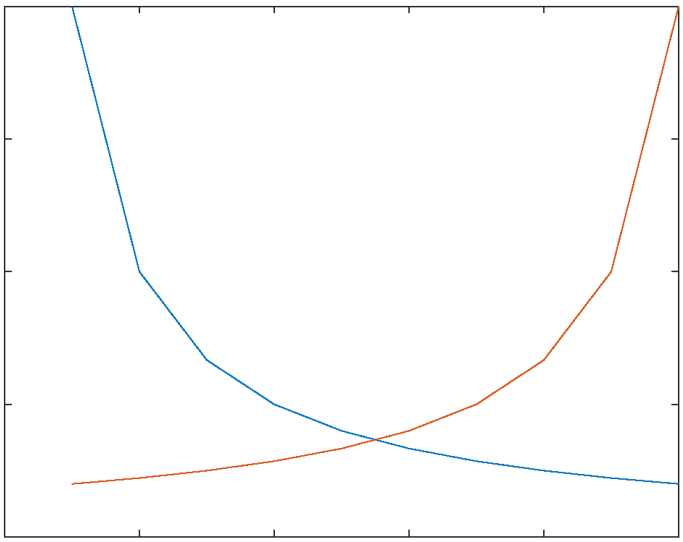

Figure 3.5: Plots of the amortized time complexities of $Push$ (blue) and Pop (red) parameterized by τ, where $l = 10$ and d is an arbitrary constant.

This completes our analysis of Algorithm 2.

CHAPTER 4

GENERALIZATIONS AND EXTENSIONS

4.1 Relaxed Priority Queues

Relaxed stacks and queues can be thought of as special cases of *relaxed priority queues*, which have operations $Insert$ and $ExtractMax$ (the one that is relaxed) and whose elements are ordered from the top in decreasing priority. We give the definition of an unrelaxed Priority Queue based on that of the unrelaxed Stack, Definition 1.

Definition 17. *A* Priority Queue *over a set of values V is a data type with two operations:*

- $Insert(val, -), val \in V$,
- $ExtractMax(-, val), val \in V \cup \{\perp\}$.

A Priority Queue provides a priority function $P : V \to S$, *where S is defined as a well-ordered set such that P is bijective. Elements are timestamped as they are added, so a Priority Queue also provides a function $t : V \to \mathbb{R}_{>0}$ that can be used to determine the timestamp when each element was $Inserted$.*

Given a sequence ρ of instances of operations on a Stack, define the following terms:

- *An $Insert(val, -)$ is* matched *if there exists a $ExtractMax(-, val)$ in ρ. If this is the case, $Insert(val, -)$ is said to* match *$ExtractMax(-, val)$ in ρ.*
- *An $ExtractMax(-, val)$ in ρ is* headmost *if there exists an $Insert(val, -)$ in ρ such that for every unmatched $Insert(val', -)$ in ρ, $P(val) > P(val')$.*
- *An $ExtractMax(-, val)$ in ρ is* resolved *if it is either headmost or succeeded by a headmost $ExtractMax$ instance.*

  A matched $Insert(val, -)$ in ρ is resolved *if some $ExtractMax(-, val)$ it matches is resolved.*

- *The* lateness *of ρ, $\lambda(\rho)$, is the number of unresolved $ExtractMax$ instances in ρ.*
- *An $ExtractMax(-, val)$ in ρ is* own-top resolved *if it is either headmost or succeeded by a headmost $ExtractMax(-, val')$ such that*
  - *$t(val) > t(val')$, and*
  - *there is no $Insert(val'', -)$ such that $t(val) > t(val'') > t(val')$.*
- *The* current-top lateness *of ρ, $\gamma(\rho)$, is the number of own-top unresolved $ExtractMax$ instances succeeding the unmatched $Insert$ in ρ whose value has the latest timestamp.*

A sequence of operation instances is legal iff it satisfies the following conditions:

(C1) The empty sequence is legal.

(C2) If ρ is a legal sequence of operation instances, then $\rho \cdot Insert(val, -)$ is legal iff there is no $Insert(val, -)$ already in ρ.

(C3) If ρ is a legal sequence of operation instances, then $\rho \cdot ExtractMax(-, val)$, $val \neq \perp$ is legal iff $ExtractMax(-, val)$ is headmost.
$\rho \cdot ExtractMax(-, \perp)$ is legal iff every instance of $Insert$ in ρ is matched.

By methodically replacing corresponding operations and terms, analogous definitions (e.g. Out-of-Order k-Relaxed Priority Queue, Lateness k-Relaxed Priority Queue, etc.) can be stated for every type of relaxed stack presented in Section 2. The ensuing discussion assumes that these definitions have been implicitly stated.

4.1.1 Reductions

Relaxed stacks are in fact relaxed priority queues in which the priority of elements is determined by how recently they were $Push$ed, where we consider the respective analogous operations to be $Push - Insert$ and $Pop - ExtractMax$. This means that relaxed priority queues behave just like relaxed stacks with the exception that $Push$ed elements can be

added at the head, the tail, or between any existing elements in the data structure depending on their priority.

Any algorithm implementing any type of relaxed stack given in Section 3 or Appendix A can be *reduced* to an algorithm implementing the analogous type of relaxed priority queue with identical performance, where *reducibility* is given in the sense that the distributed time complexity in terms of our model is unchanged. The procedure is detailed in the following theorem.

Theorem 19. *If the following changes are applied to any existing algorithm that implements a relaxed stack:*

- *the local variables are defined identically as in the beginning of Section 3, although names may be changed as appropriate;*
- *the* topmost *and* bottommost *elements now respectively refer to the elements of* highest priority *and the elements of* lowest priority *in all instances, except that*
  - *wherever we previously considered the* topmost volatile *elements, we now instead use volatile elements having the* latest timestamp*, defining the new operations* $latestBySet()$ *and* $latestSetBySet()$ *on the local priority queue for this purpose;*
- *the definitions of the terms and the labeling, uniformity, and headmost invariants are the same as before, in the algorithms where they are applicable;*
- *when an* $Insert$ *takes place,*
  - *if the* $Inserted$ *element is placed either in the safe region or at a higher priority than some element in the safe region, then the procedure followed in the* $Insert$*'s local execution remains the same (the* $Push$ *operation in the relaxed stack algorithms falls within this category);*

- *if it is placed at a lower priority than all the elements in the safe region but still within the set of elements legal to return, then we follow the more straightfoward procedure of assigning it a dummy label and unlabeling the element moved out of the legal set in order to maintain the labeling invariant, while the safe region and hence the legality of elements and all the other invariants are preserved;*
- *otherwise, if it is outside the set of legal elements, then all the legal elements are preserved, so no labels need to be changed;*

- *for the relaxed $ExtractMax$ operation, the executions in the messages it passes and its local execution remain the same;*

then the new algorithm correctly implements the corresponding relaxed priority queue, and furthermore, the relaxed priority queue shares the same worst-case and amortized time complexities and lower bounds as its analogous relaxed stack. □

Proof. If we combine the reasoning associated with the augmented procedures for $Insert$ in addition to the fact that the relaxed $ExtractMax$ operation behaves identically to the Pop operation, then the reasoning for the proof of correctness represented by the statements and proofs of each lemma and theorem will be identical. The performance proofs also remain the same because the changes are all accounted for during local execution of operation instances. □

As an example, the reduction from Theorem 19 applied to the Out-of-Order k-Relaxed Stack algorithm (Algorithm 2) results in the Out-of-Order k-Relaxed Priority Queue algorithm (described in Appendix B as Algorithm 7).

The currently known optimal bounds are summarized comprehensively in Tables 4.1 and 4.2.

Table 4.1: Bounds on $Push$/$Insert$ Time Complexity*

	Worst-Case Cost		Amortized Cost	
	Lower Bound	Upper Bound	Lower Bound	Upper Bound
Unrelaxed	$(1-\frac{1}{n})u$ [2]	ϵ [2]	?	ϵ †
Out-of-Order, Restricted, Windowed	?	$\max\{2d+(1-\tau_c)\epsilon, \epsilon\}$	?	$\max\{\frac{2d}{\tau_c}, \epsilon\}$
Lateness	?	ϵ	?	ϵ

Table 4.2: Bounds on Pop/$ExtractMax$ Time Complexity

	Worst-Case Cost		Amortized Cost	
	Lower Bound	Upper Bound	Lower Bound	Upper Bound
Unrelaxed	$d+\min\{\epsilon, u, \frac{d}{3}\}$ [2]	$d+\epsilon$ [2]	$d(1-\frac{1}{n})$ [11]	$d+\epsilon$
Out-of-Order	$\frac{d}{l}, k < n^2$ ‡	$\max\{d+\epsilon$[2], $2d+(\tau-l)\epsilon, \epsilon\}$	$\frac{d}{l}, k < n^2$ [11]	$\max\{\frac{2d}{l-\tau+1}, \epsilon\}$
Lateness	d [11]	$d+\epsilon$ [2]	$> \frac{d}{\lceil k/n \rceil}$ ††	$\frac{d+u-\epsilon}{l_s}+\epsilon$
Restricted, Windowed	d [11]	$d+\epsilon$ [2]	$> \frac{d}{\lceil k/n \rceil}$ [11]	$\frac{2d-\epsilon}{\min\{l_s, l-\tau\}}+\epsilon$
Per-Top	d [11]	$d+\epsilon$ [2]	?	$d+\epsilon$
Stuttering	d [11]	$d+\epsilon$ [2]	?	$d+\epsilon$

*Assumes the bounds on Pop/$ExtractMax$ given in Table 4.2.

†Upper bound on worst-case cost is an upper bound on amortized cost.

‡Lower bound on amortized cost is a lower bound on worst-case cost.

††The lower bound proof is the same as for the Restricted k-Relaxed Stack given in [11]. (The same proof contains a minor error in that $\lceil k/n \rceil$ should be used in place of l.)

4.1.2 Relaxed Queues as a Special Case

Because relaxed priority queues also generalize relaxed queues, where we consider the respective analogous operations to be $Enqueue - Insert$ and $Dequeue - ExtractMax$, we can deduce that the same upper and lower bounds apply for relaxed queues by extension, although it is obvious that they may not necessarily be the optimal upper and lower bounds [11] due to the specific properties of relaxed queues. In particular, we give a nontrivially-bounded algorithm implementing the Stuttering k-Relaxed Queue in Appendix A.4.

Because one property of relaxed queues is that the topmost element is changed only when it is removed, in the definitions where it applies, all unresolved elements become resolved whenever the topmost element is changed. Hence, the Lateness and Per-Top Lateness relaxations both equivalently correspond to the Lateness k-Relaxed Queue [10], and all four of the Restricted, Windowed, Per-Top Restricted, and Per-Top Windowed relaxations equivalently correspond to the Restricted Out-of-Order k-Relaxed Queue [10], which we refer to interchangeably as the *Restricted k-Relaxed Queue* or the *Windowed k-Relaxed Queue*.

In the same way that the Restricted k-Relaxed Priority Queue algorithm (reduced from Algorithm 5) provides an upper bound only for heavily loaded runs whereas the Windowed k-Relaxed Priority Queue algorithm (reduced from Algorithm 4) does so for any runs in general, a new Restricted k-Relaxed Queue algorithm derived from the Windowed k-Relaxed Priority Queue algorithm exhibits a similar additional performance improvement. The Restricted k-Relaxed Queue algorithm will hereafter refer to this derivation.

We give three theorems which illustrate further conditional properties of relaxed priority queue algorithms that apply unconditionally to relaxed queues, leading to their improved performance. It is worthwhile to mention that the Lateness Out-of-Order k-Relaxed Priority Queue algorithm (reduced from Algorithm 3) satisfies these theorems as well, which is why it exhibits the same performance improvements as the relaxed queue algorithms.

Definition 18. *An $Insert$ is* safe *if it is guaranteed not to move any element out of the set of safe elements in each local priority queue at local execution.*

Theorem 20. *Safe $Inserts$ are always permitted to be fast, taking ϵ time to respond, without changing the response time of $ExtractMaxes$, unless they return $\perp$ before local execution in the implementations of relaxed priority queues where returning $\perp$ depends on the size of the local stack.* □

Proof. $ExtractMax$es which are invoked after a safe $Insert$'s invocation but return an element in the local stack before its local execution will still behave legally as when no such safe $Insert$ is invoked – because the $ExtractMax$ necessarily returned before its local execution, the returned element is safe by definition, and will not be made unsafe by the safe $Insert$. Thus, allowing the $Insert$ to respond at the earliest time possible does not require an increase in the response time of such $ExtractMax$es. □

In particular, if the $Insert$ed element is placed below the safe region but is still among the set of elements legal to return, then the necessary changes in labeling to maintain the labeling invariant will not be made too soon or too late since every operation instance reassigns labels only at its local execution, which follows the same linearized ordering as invocation.

On the other hand, if a $ExtractMax$ may return $\perp$ before local execution, then since the number of fast $Insert$s that may be invoked before the $ExtractMax$ is invoked but locally executed after it returns may be much larger than k (depending on the value of d relative to ϵ), the sequence obtained from linearizing these operation instances by invocation time may actually be illegal, if the definition of the relaxed priority queue requires that the number of unmatched $Insert$s is smaller than some number at most k. This means that in implementations of such relaxed priority queues, all $ExtractMax$es returning $\perp$ may only do so at local execution, so no $ExtractMax$es returning $\perp$ are fast. We will see that this impacts the performance of the Out-of-Order k-Relaxed Queue and the Restricted

k-Relaxed Queue but not any of the others.

Another caveat is that an $Insert$ can only be fast if it is possible to decide that it is safe within ϵ time of invocation, during which the state of the invoking process' local priority queue may differ from that during the time of the $Insert$'s local execution. This is not an issue for relaxed queues, as $Enqueue$ operations add elements to the opposite end of the data structure from where they will be removed and hence will not affect the legality of existing elements according to any relaxed queue definition, so Theorem 20 states that every $Enqueue$ is predetermined to be safe.

Theorem 21. *The $d + u$ minimum restriction on the pending delay that originates from Algorithms 2, and 4, and 5 does not apply if $Inserts$ are safe.* □

Proof. If the condition holds, then the $t_{volatile}$ field and $remsIncomingSeen$ ($popsInPending$ in relaxed stacks) counter become extraneous because their use is contingent on the possibility that elements may be relabeled, which in turn is not necessary if elements never leave and re-enter the safe region, so the related actions that take place in messages passed by $ExtractMax$es can be safely ignored. Thus, the restriction of $2d$ required by the message passing on the time between invocation and local execution of operation instances can be removed. □

The new pending delay for Algorithm 2 becomes $u + \epsilon$ so that operation instances still execute locally in timestamp order by Lemma 1. The new pending delay for Algorithms 4 and 5 becomes $2u$, as is required by the headmost invariant from the proof of correctness of Algorithm 3.

Theorem 22. *If $Inserts$ are safe, then the safe elements include all elements that are legal to return by locally executing $Inserts$.* □

Proof. Since elements are never relabeled, if an element is currently legal for a locally executing $ExtractMax$ to return, then it is also legal to return by any $ExtractMax$ that will locally execute in the future, and hence we conclude that the element must be safe.

From the proof of Theorem 21, there are no volatile or lower regions, so the definitions of the labeling invariants state in this setting that labeled elements include only elements that are legal for a locally executing $ExtractMax$ to return. □

Hence, in relaxed queue algorithms the uniformity invariant applies to the entire set of labeled elements, which is larger than the safe region as defined in the relaxed stack algorithms, and this leads to a further performance improvement.

Because there is no limit to the number of mutually active fast $Dequeue$s and there is no contention region, the algorithms and their time complexities for relaxed queues are independent of the τ parameter.

In summary, we may reduce from relaxed stack algorithms to relaxed priority queue algorithms and then apply the optimizations given in these theorems to the algorithms if they are restricted to specifically implement relaxed queues. The reasoning for the corresponding performance proofs for the Lateness k-Relaxed Stack algorithm in Section 3 is identical for the optimized Lateness k-Relaxed Queue algorithm, allowing us to obtain new upper bounds for this type of relaxed queue.

However, we recall that if Theorem 20's optimization is applied to the Out-of-Order k-Relaxed Queue and Restricted k-Relaxed Queue algorithms, then any $Dequeue$ returning $\perp$ requires $d + \epsilon$ time, and this new restriction changes the performance proofs for these data structures.

We begin by introducing a definition describing runs in which the size of the local stack for each process is at least k before every $ExtractMax$, so that $\tau_p = \tau_c = \tau$.

Definition 19. *A run R is* heavily loaded *if, for some linearization π of R, every prefix of π which is immediately followed by an instance of $ExtractMax$ has at least k unmatched $Insert$.*

It turns out that for heavily loaded runs, the amortized complexities for these data structures still follow the same proofs as before, so we obtain results for this specific case. The

amortized time complexities for non-heavily loaded runs require new proofs, however. For this purpose, we introduce a modified form of the l parameter which is the same as before in heavily loaded runs, but gives the number of elements labeled with each process id in each process' local stack at any time.

Definition 20. *At any process p_i, the* effective-l*, l_e, and the* effective-l_s*, l_{se}, are defined as*

$$l_e := \begin{cases} l, & lPQ.size() \geq k \\ \lfloor I/n \rfloor, & lPQ.size() < k \end{cases} \quad \text{and} \quad l_{se} := \begin{cases} l_s, & lPQ.size() \geq k \\ \lfloor I/n \rfloor, & lPQ.size() < k \end{cases}$$

where I is the number of $Inserts$ that executed locally in p_i after the local execution of the previous $ExtractMax$.

Theorem 23. *For a complete, admissible execution of the derived Out-of-Order k-Relaxed Queue algorithm, the amortized time complexity of $Dequeues$ at any process p_i during a time interval in which its effective-l is given by l_e is at most $d/(l_e + 1) + \epsilon$.* □

Proof. The number of elements labeled p_i during the given time interval must be at least l_e, because if the size of p_i's local stack is less than k and I consecutive $Enqueue$s execute locally, then the $Enqueue$s must have labeled at least l_e elements with each process' id.

Thus, if a slow $Dequeue$ is invoked in p_i, it must have followed l_e fast $Dequeue$s, which returned the elements labeled p_i and marked them $dequeued$. Since each fast $Dequeue$ takes ϵ time and the slow $Dequeue$ takes $d + \epsilon$ time, the average cost of each repetition of a pattern of l_e fast $Dequeue$s and one slow $Dequeue$ is

$$\frac{l_e\epsilon + (d + \epsilon)}{l_e + 1} = \frac{d + (l_e + 1)\epsilon}{l_e + 1} = \frac{d}{l_e + 1} + \epsilon.$$

The cost of any prefix of the infinite repetition of this pattern is bounded above by the maximum average cost of a single copy of this pattern, since prefixes ending with a slow $Dequeue$ which ends a pattern have the highest average cost.

Because this provides an upper bound on the cost of $Dequeue$s at any process p_i, it is also an upper bound on the average cost of $Dequeue$s at all processes; hence, it is the amortized cost. □

Theorem 24. *For a complete, admissible execution of the derived Restricted k-Relaxed Queue algorithm, the amortized time complexity of $Dequeue$s at any process p_i during a time interval in which its effective-l is given by l_{se} is at most $(d + u - \epsilon)/l_{se} + \epsilon$ if $l_{se} > 0$, or $d + u$ otherwise.* □

Proof. The proof is similar as in the previous theorem, but the pending delay is now $2u$ causing the response time of slow $Dequeue$s to become $(d - u) + 2u = d + u$; additionally, the number of elements labeled p_i that can be returned by a fast $Dequeue$ is $l_{se} - 1$; the topmost is excluded, since by the headmost invariant it cannot be returned by a fast $Dequeue$. □

Table 4.3: Bounds on $Dequeue$ Time Complexity in Heavily-Loaded Runs Only**

	Worst-Case Cost		Amortized Cost	
	Lower Bound	Upper Bound	Lower Bound	Upper Bound
Unrelaxed	$d + \min\{\epsilon, u, \frac{d}{3}\}$[2]	$d + \epsilon$ [2]	$d(1 - \frac{1}{n})$ [11]	$d + \epsilon$
Out-of-Order	$\frac{d}{l}, k < n^2$	$\max\{d + (1 - l)\epsilon, \epsilon\}$	$\frac{d}{l}, k < n^2$ [11]	$\max\{\frac{d+\epsilon}{l+1}, \epsilon\}$
Lateness	d [11]	$d + \epsilon$ [2]	$> \frac{d}{\lceil k/n \rceil}$ ††	$\frac{d+u-\epsilon}{l_s} + \epsilon$
Restricted	d [11]	$d + \epsilon$ [2]	$> \frac{d}{\lceil k/n \rceil}$ [11]	$\frac{d+u-\epsilon}{l_s} + \epsilon$
Stuttering	d [11]	$d + \epsilon$ [2]	?	$\frac{d}{l_s+1} + \epsilon$

††The lower bound proof is the same as for the Restricted k-Relaxed Stack given in [11]. (The same proof contains a minor error in that $\lceil k/n \rceil$ should be used in place of l.)

**Assumes an upper bound of ϵ on $Enqueue$, which changes the Out-of-Order and Restricted results.

Table 4.4: Bounds on $Dequeue$ Time Complexity**

	Worst-Case Cost		Amortized Cost	
	Lower Bound	Upper Bound	Lower Bound	Upper Bound
Unrelaxed	$d + \min\{\epsilon, u, \frac{d}{3}\}$[2]	$d + \epsilon$ [2]	$d(1 - \frac{1}{n})$ [11]	$d + \epsilon$
Out-of-Order	$\frac{d}{l}, k < n^2$	$d + \epsilon$ [2]	$\frac{d}{l}, k < n^2$ [11]	$\frac{d}{l_e+1} + \epsilon$
Lateness	d [11]	$d + \epsilon$ [2]	$> \frac{d}{\lceil k/n \rceil}$	$\frac{d+u-\epsilon}{l_s} + \epsilon$
Restricted	d [11]	$d + \epsilon$ [2]	$> \frac{d}{\lceil k/n \rceil}$ [11]	$\frac{d+u-\epsilon}{l_{se}} + \epsilon$
Stuttering	d [11]	$d + \epsilon$ [2]	?	$d + \epsilon$

The upper bounds on time complexities for relaxed queues are aggregated in Tables 4.3 and 4.4 (no table is given for $Enqueue$ since it is assumed that the upper bounds are all ϵ), which illustrate the following new results:

- The upper bounds are improved on $Dequeue$ operations in heavily loaded runs of Out-of-Order k-Relaxed Queues, originating from slow Pops being able to respond in less than $d + \epsilon$ time.
- A correct algorithm implementing the Lateness k-Relaxed Queue is (implicitly) derived.
- The derived Restricted k-Relaxed Queue algorithm improves the previous upper bound by nearly a factor of 2, and also guarantees an upper bound on non-heavily loaded runs.

Finally, if we decide to apply the same derivation as before for the Out-of-Order k-Relaxed Queue and Restricted k-Relaxed Queue algorithms, but without Theorem 20's optimizations this time, then repeating the original performance proofs gives a few new results that apply generally (not just for heavily loaded runs). These are given in Table 4.5.

Table 4.5: Upper Bounds without Theorem 20's Optimizations in the General Case

	Worst-Case Cost		Amortized Cost	
	$Enqueue$	$Dequeue$	$Enqueue$	$Dequeue$
Out-of-Order	$\max\{d + (2 - \tau_c)\epsilon, \epsilon\}$	$\max\{d + (\tau - l + 1)\epsilon, \epsilon\}$	$\max\{\frac{d+\epsilon}{\tau_c}, \epsilon\}$	$\frac{d+\epsilon}{l-\tau+1}$
Restricted	$\max\{d + (2 - \tau_c)\epsilon, \epsilon\}$	$d + \epsilon$	$\max\{\frac{d+\epsilon}{\tau_c}, \epsilon\}$	$\frac{d+u-\epsilon}{\min\{l_s, l-\tau\}} + \epsilon$

For the purpose of completeness, the completely optimized (including Theorem 20) algorithms derived from the original relaxed stack algorithms are provided as Algorithms 8, 9, and 10 in Appendix B, simplified by having the functionally extraneous pseudocode removed.

In the Out-of-Order k-Relaxed Queue algorithm, the reason for $reservedQueues[]$ not being needed is that new elements are added and labeled in the same ordering as the queue data structure itself.

In both of the Out-of-Order k-Relaxed Queue and Restricted k-Relaxed Queue algorithms, line 5 has been modified to prevent a fast $Dequeue$ from returning $\perp$. $localFastPopsInvoked$ and $localFastPopsExecuted$ become extraneous in the latter because its only purpose was to limit the number of consective fast Pops that returned $\perp$. For the former in particular, lines 26-28 have been added to ensure that a slow $Dequeue$ that would have previously returned $\perp$ instead continues for $d + \epsilon$ time before response, while for the latter, slow $Dequeue$s always respond in $d + \epsilon$ time.

CHAPTER 5
CONCLUSION

From the starting point of relaxed stacks, we have fully characterized efficient algorithms for linearizable relaxed priority queues under a particular distributed system model. One way of conceptually surveying the relationships between the relaxed ordered data types in hindsight is that relaxed stacks are the "worst case" while relaxed queues are the "best case" of relaxed priority queues.

We intend to implement these data types in software with the help of existing libraries such as Open MPI, and compare them to existing benchmarks as well as release them under an open-source license. Relaxed ordered data types have previously been implemented in shared memory [10]; relaxed priority queues in particular have also seen implementations under different models and been analyzed empirically [13], [14]. From a broader perspective, we see an ongoing trend toward the loosening of the semantics of data types in a reasonable manner in order to improve their performance and scalability in the most general types of distributed systems. The work done so far in this paper serves to propel the theoretical aspects underlying this research direction.

Relaxed priority queues have a rich set of applications that we would like to explore, looking forward. For example, *discrete event simulators*, which are used to model the systems underlying scenarios as diverse as chemical processing, automotive manufacturing, and network traffic monitoring, can be implemented efficiently using priority queues. It is possible that if the data is sufficiently well-behaved, then relaxed priority queues can be used in place of priority queues in many more performance-critical settings where computation is distributed across multiple clusters.

Appendices

APPENDIX A

ADDITIONAL ALGORITHMS AND UPPER BOUNDS

A.1 Lateness Relaxed Stacks

Algorithm 3 implements the Lateness k-Relaxed Stack (Definition 3). The algorithm actually enforces the slightly stronger lateness condition that

$$\lambda(\rho) \leq l_s n = \left(\left\lceil \frac{k}{n} \right\rceil - 1 \right) n = \begin{cases} k - n, & n \mid k \\ \lfloor k/n \rfloor \cdot n, & n \nmid k \end{cases} \tag{A.1}$$

rather than $\lambda(\rho) < k$ as in Definition 3, given that ρ denotes the linearization of operation instances.

$localPopsInvoked$ and $localPopsExecuted$ are newly utilized local variables intended to ensure the above lateness condition. We show that $localPopsInvoked$ and $localPopsExecuted$ are respectively the number of unresolved Pops invoked and the number of unresolved Pops executed locally by the process. We can see from the algorithm that $localPopsExecuted$ never exceeds $localPopsInvoked$, which is capped at l_s in each process.

The *labeling invariant* in this algorithm states that every element in the local stack is labeled, because by the specification given by Definition 3, every element is legal to return as long as $\lambda(\rho) < k$. Labels are never changed after they are assigned, so there is no contention region, and all $Push$es are permitted to fast, taking ϵ time; thus, the algorithm is independent of τ and is designed to optimize the performance of Pops.

The *headmost invariant* states that in each process p_i, the topmost non-$popped$ element labeled p_i, and only that element, has $canPopByLabel = false$. $canPopByLabel$ is also a newly utilized local variable which must be $true$ in order for the element to be returned

by $popByLabel()$ or $peekByLabel()$.

When $localPopsInvoked = l_s$ in some process, the next Pop to be invoked will be slow. The headmost invariant ensures that the very first slow Pop, pop_1, to execute locally always returns and removes the top element in the local stack (since it is the topmost element assigned some label), resolving all the previously locally executed Pops. Any slow Pops mutually active with pop_1 will then have $localPopsExecuted = 0$, so they are free to return and remove a non-top element.

The first locally executing slow Pop not mutually active with pop_1, pop_2, will return and remove the topmost non-$popped$ element, $elem$. We will see in the proof of correctness that because it is not $popped$, $elem$ cannot have already been returned by a fast Pop, so legality is maintained. Additionally, let p_i be the process that invoked pop_2. If $elem$ is not the top element, then the top element, being $popped$, must have been returned by a fast Pop whose timestamp will be earlier than that of the next Pop invoked by p_i. That fast Pop resolves the previous Pops, and thus p_i is permitted to invoke l_s more fast Pops.

The reasoning above for pop_2 also holds for any later-timestamp slow Pop not mutually active with pop_2, which we call pop_3, and so on by induction.

Algorithm 3 Pseudocode for each process p_i implementing a Lateness k-Relaxed Stack, where $k \geq n$.

```
1:  HandleEvent PUSH(val)
2:      send (push, val, ⟨localTime, i⟩) to all
3:      setTimer(ϵ, ⟨push, null, ⟨localTime, i⟩⟩, respond)
4:  HandleEvent POP
5:      if localPopsInvoked < l_s then
6:          localPopsInvoked++
7:          let ret := lStack.peekByLabel(p_i)
8:          ret.popped := true
9:          send (pop_f, ret, ⟨localTime, i⟩) to all
10:         setTimer(ϵ, ⟨pop_f, ret, null⟩, respond)
11:     else send (pop_s, null, ⟨localTime, i⟩) to all
12: HandleEvent RECEIVE (op, val, ts) FROM p_j
13:     Pending.insert(⟨op, val, ts⟩)
```

14: $setTimer(2u, \langle op, val, ts\rangle, execute)$

15: **if** $op == pop\_f$ **then** $val.popped := true$

16: **HandleEvent** EXPIRETIMER($\langle op, val, ts\rangle, respond$)

17: **if** $op == pop\_f$ **then return** val

18: **else return** ACK

19: **HandleEvent** EXPIRETIMER($\langle op, val, ts\rangle, execute$)

20: **while** $ts \geq Pending.min()$ **do**

21: $\langle op', val', ts'\rangle := Pending.extractMin()$

22: $executeLocally(op', val', ts')$

23: $cancelTimer(\langle op', val', ts'\rangle, execute)$

24: **function** EXECUTELOCALLY($op, val, \langle *, j\rangle$)

25: **if** $op == push$ **then**

26: $lStack.topAvailableByLabel(p_i).canPopByLabel := true$

27: $lStack.push(val)$

28: $lStack.label(\text{argmin}_{p_h} |lStack.setByLabel(p_h)|, val)$

29: $lStack.topAvailableByLabel(p_i).canPopByLabel := false$

30: **else**

31: **if** $op == pop\_f$ **then**

32: **if** $j == i$ **then** $localPopsExecuted{+}{+}$

33: **if** $val == lStack.getFromTop(1)$ **then**

34: $localPopsInvoked \mathrel{-}= localPopsExecuted$

35: $localPopsExecuted := 0$

36: $lStack.remove(val)$

37: **else**

38: **if** $localPopsExecuted == l_s$ **then**

39: $localPopsInvoked \mathrel{-}= localPopsExecuted$

40: $localPopsExecuted := 0$

41: let $ret := lStack.pop()$

42: **if** $ret.label == p_i$ **then**

43: $lStack.peekByLabel(p_i).canPopByLabel := false$

44: **else**

45: **if** $j == i$ **then**

46: $localPopsInvoked{+}{+}$

47: $localPopsExecuted{+}{+}$

48: let $ret := lStack.popByLabel(p_j)$

49: **if** $j == i$ **then return** ret

A.1.1 Correctness

Let $ts(op)$ denote the timestamp associated with an operation instance op given as the first argument in line 2, 9, or 11.

Construction 2. *Define the permutation π of operation instances in a complete, admissible run of Algorithm 3 as the order given by sorting by $ts(op)$ for each operation instance op.*

In Algorithm 3, the pending delay is $2u$ (line 14); we show that Lemma 1 holds.

Lemma 25. *Each process running Algorithm 3 locally executes all $Pushes$ and $Pops$ in timestamp order.*

Proof. Since $2u > u + \epsilon$, Lemma 1 holds for Algorithm 2. □

The proof of the following fact is the same as in Algorithm 2.

Lemma 26. *π respects real-time order of non-overlapping operation instances.*

In the remainder of the proof of correctness, we give a useful lemma and then show inductively that the local execution of each operation type satisfies the invariants defined thus far and maintains a legal linearization, thereby implementing a Lateness k-Relaxed Stack according to Definition 2.

Lemma 27. *In each process, $localPopsInvoked \geq localPopsExecuted$.*

Proof. $localPopsInvoked$ and $localPopsExecuted$ are both initialized to 0.

$localPopsExecuted$ is incremented only at the local execution of a Pop (lines 32 and 47). Whenever $localPopsExecuted$ is incremented, $localPopsInvoked$ has also been incremented at either the invocation (line 6) or local execution (line 46) of the same Pop.

Also, whenever $localPopsInvoked$ is decreased (lines 34 and 39), $localPopsExecuted$ is decreased by the same amount (lines 35 and 40). □

Lemma 28. *Every Pop returns a unique element.*

Proof. Fast Pops as well as slow Pops when $localPopsExecuted < l_s$ return only elements sharing the same label as the invoking process, and fast Pops set their returned elements' $popped$ fields to $true$ (the aforementioned Pops rely on the $lStack$ functions $peekByLabel()$ and $popByLabel()$), while slow Pops remove returned elements from the same process' local stack.

The only time processes return elements not necessarily labeled with their own ids is during slow Pops when $localPopsExecuted = l_s$ (line 38). Such a slow Pop, pop_2, cannot be mutually active with a preceding slow Pop in which $localPopsExecuted = l_s$, because the latter would have set $localPopsExecuted := 0$ in all processes, and no fast Pops could have been invoked in between. If there is no preceding slow Pop not mutually active with pop_2, then assuming the headmost invariant holds, pop_2 returns an element with $canPopByLabel = false$, which cannot have been returned by any of the Pops mentioned above.

Otherwise, there exists a preceding slow Pop not mutually active with pop_2, the latest of which we call pop_1. The local execution of pop_1 removes some element with $canPopByLabel = false$, and sets $canPopByLabel = false$ for the topmost remaining non-$popped$ element $elem$ sharing the same label, which we call p_i. Any $popped$ elements above $elem$ labeled p_i, if they exist, must have been claimed by fast Pops invoked earlier in real time than the local execution of pop_1 in p_i; let pop_f denote the latest such fast Pop. Because local execution differs in real time between processes by at most u, pop_2 may have been invoked in its own process earlier than pop_f by any amount less than u in real time. However, when the message sent by pop_f is received in any process, which takes at most d real time and hence takes place less than $d + u$ real time after pop_2 is invoked, it marks the returned element $popped$. pop_2 executes locally in each process at a later real time, $(d - u) + 2u = d + u$ after pop_2 is invoked at the earliest.

Because pop_2 removes the topmost non-$popped$ element at local execution (line 41), it will not return the same element as pop_f or any of the earlier fast Pops, if they exist. □

Theorem 29. *For any execution of Algorithm 3, the permutation π given by Definition 2 is legal by the specification of a Lateness k-Relaxed Stack. Thus, Algorithm 3 is a correct implementation of a Lateness k-Relaxed Stack.* □

Proof. We consider local execution by operation type, and show that for every sequence of operation instances invoked by the processes, the algorithm (a) generates return values such that π is legal, and also maintains (b) the labeling invariant and (c) the headmost invariant. (b) and (c) are demonstrated by induction on the local stack, with the base case being that the invariants are vacuously satisfied when the stack is empty.

$Push$:

(a) As long as every value is unique, *(C2)* in Definition 1 is satisfied and hence any $Push$ is legal.

(b) If every element is labeled, then since the newly $Push$ed element is labeled also (line 28), the labeling invariant holds.

(c) In each process p_i, the topmost non-$popped$ element labeled p_i before the $Push$ has $canPopByLabel = false$, assuming the headmost invariant holds. Following the $Push$, the new topmost non-$popped$ element labeled p_i has $canPopByLabel$ set to $false$ (line 29). If the previous topmost non-$popped$ element labeled p_i was distinct, then it has $canPopByLabel$ set to $true$ (line 26).

Pop:

(a) We use the fact that every process executes operation instances in linearized order. First, by Lemma 28, Pops return unique elements, so they match unmatched $Push$es.

We now show that Pops return values maintaining the legality of the linearization. An invoked Pop is fast when $localPopsInvoked < l_s$ (line 5), and as a result it increments $localPopsInvoked$ (line 6) so that the sum of $localPopsInvoked$ among all processes is at most $l_s n < k$. Any fast Pop also increments $localPopsExecuted$ at local execution. On the other hand, an invoked Pop, pop, is slow when

$localPopsInvoked = l_s$. This case is covered as follows.

If $localPopsExecuted = l_s$ at *pop*'s local execution, then *pop* is not mutually active with a preceding slow Pop in which $localPopsExecuted = l_s$, because such a slow Pop would set $localPopsExecuted := 0$ in all processes (line 40). *pop* returns the topmost non-*popped* element $elem$, and either $elem$ is the top element or a fast Pop returned the top element; either case resolves all the previously locally executed Pops, so it is justified to decrease $localPopsInvoked$ and $localPopExecuted$ by the same amount such that $localPopExecuted = 0$.

Otherwise, if $localPopsExecuted \neq l_s$ at *pop*'s local execution, then $localPopsExecuted = 0$ because *pop* must be mutually active with a preceding slow Pop in which $localPopsExecuted = l_s$. By the previous paragraph, *pop* must have resolved all the unresolved Pops invoked by the same process as *pop*, so *pop* may return a non-top element (line 48) before incrementing both $localPopsInvoked$ and $localPopsExecuted$.

Because the number of unresolved instances is always less than k, Pops maintain *(C3)* in Definition 3.

(b) Because elements are never unlabeled, the labeling invariant still holds.

(c) By the headmost invariant, only a locally executing slow Pop when $localPopsExecuted = l_s$ removes an element with $canPopByLabel = false$ in any process. In the same process, the Pop sets the new topmost non-*popped* element's $canPopByLabel$ field to $false$ (line 43), maintaining the headmost invariant.

□

A.1.2 Performance

An advantage of the Lateness k-Relaxed Stack implementation is that all $Push$es are fast, which we state below.

Theorem 30. *The worst-case and amortized time complexities of $Push$ among all complete, admissible executions of Algorithm 3 is no more than ϵ.* □

Theorem 31. *The amortized time complexity of Pop among all complete, admissible executions of Algorithm 3 is no more than $(d + u - \epsilon)/l_s + \epsilon$.* □

Proof. We consider the view of a single process p_i in which slow Pops occur. The first slow Pop occurs when $localPopsInvoked$ has been incremented to l_s by the same number of fast Pops. If the slow Pop is mutually active with another slow Pop in whose process $localPopsExecuted = l_s$, then before the slow Pop, $localPopsExecuted = 0$ in p_i and afterward, $localPopsInvoked$ and $localPopsExecuted$ become 1 (lines 46-47). In any other case, $localPopsInvoked$ and $localPopsExecuted$ become 0 in p_i. This means at least $l_s - 1$ subsequent Pops, if they exist, will be fast, and this pattern repeats.

Since each fast Pop takes ϵ time and the slow Pop takes $d + u$ time, the average cost of each repetition of the pattern of $l_s - 1$ fast Pops and 1 slow Pop is

$$\frac{(l_s - 1)\epsilon + (d + u)}{l_s} = \frac{d + u - \epsilon + l_s\epsilon}{l_s} = \frac{d + u - \epsilon}{l_s} + \epsilon.$$

The cost of any prefix of the infinite repetition of this pattern is bounded above by the maximum average cost of a single copy of this pattern, since prefixes ending with a slow $Push$ which ends a pattern have the highest average cost.

Because this provides an upper bound on the cost of $Push$es at any process p_i, it is also an upper bound on the average cost of $Push$es at all processes; hence, it is the amortized cost. □

A.2 Windowed Relaxed Stacks

Algorithm 4 implements the Windowed k-Relaxed Stack algorithm. Because it is nearly completely a combination of the Out-of-Order k-Relaxed Stack algorithm (Algorithm 2)

and the Lateness k-Relaxed Stack algorithm (Algorithm 3), it is recommended that Algorithm 4 be compared side-by-side with those algorithms.

Having $unresolved = true$ signifies that an element has already been removed but is not yet resolved, which defines a class of elements that is essential to Definition 6.

A slow Pop always returns and removes the top element at its local execution. It then proceeds to remove all the unresolved elements one-by-one while relabeling using the same method found in Algorithm 2.

Similar to Algorithm 3, this algorithm uses the $localPopsInvoked$ and $localPopsExecuted$ local variables, which here are necessary to limit the number of Pops returning $\perp$, because they qualify as unresolved Pops according to Definition 6.

Algorithm 4 Pseudocode for each process p_i implementing a Windowed k-Relaxed Stack, where $k \geq n$.

```
1:  HandleEvent PUSH(val)
2:      if localFastPushesActive < τ_c − 1 then
3:          send (push_f, val, ⟨localTime, i⟩) to all
4:          setTimer(ϵ, ⟨push_f, null, ⟨localTime, i⟩⟩, respond)
5:          localFastPushesActive++
6:      else
7:          send (push_s, val, ⟨localTime, i⟩) to all
8:          setTimer(ϵ, ⟨push_s, 1, ⟨localTime, i⟩⟩, respond)
9:  HandleEvent POP
10:     if (lStack.peekByLabel(p_i, safeRegion() ≠ ⊥ or lStack.size() < k − τn)
            and localPopsInvoked < l_s then
11:         localPopsInvoked++
12:         let ret := lStack.peekByLabel(p_i, safeRegion())
13:         ret.popped := true
14:         send (pop_f, ret, ⟨localTime, i⟩) to all
15:         setTimer(ϵ, ⟨pop_f, ret, null⟩, respond)
16:     else send (pop_s, null, ⟨localTime, i⟩) to all
17: HandleEvent RECEIVE (op, val, ts) FROM p_j
18:     Pending.insert(⟨op, val, ts⟩)
19:     setTimer(d + u, ⟨op, val, ts⟩, execute)
20:     if op == pop_f or op == pop_s then
21:         popsInPending++
```

22: **if** $op == pop\_f$ **then**
23: $val.popped := true$
24: $resetVolatility(val)$
25: **HandleEvent** EXPIRETIMER($\langle op, val, \langle *, j\rangle\rangle, respond$)
26: **if** $op == pop\_f$ **then return** val
27: **else if** $op == push\_s$ and $localFastPushesActive == \tau_c - 1$ **then**
28: $setTimer(\min\{\epsilon, 2d - (\tau_c - 1)\epsilon - val \cdot \epsilon\}, \langle push\_s, val + 1, \langle *, j\rangle\rangle, respond)$
29: **else return** ACK
30: **HandleEvent** EXPIRETIMER($\langle op, val, ts\rangle, execute$)
31: **while** $ts \geq Pending.min()$ **do**
32: $\langle op', val', ts'\rangle := Pending.extractMin()$
33: $executeLocally(op', val', ts')$
34: $cancelTimer(\langle op', val', ts'\rangle, execute)$
35: **function** EXECUTELOCALLY($op, val, \langle time, j\rangle$)
36: **if** $op == push\_f$ or $op == push\_s$ **then**
37: $lStack.topAvailableByLabel(p_i).canPopByLabel := true$
38: $lStack.push(val)$
39: **if** $lStack.size() \leq k - \tau n$ **then**
40: $lStack.label(\text{argmin}_{p_h} |lStack.setByLabel(p_h) \cap safeRegion()|, val)$
41: **else**
42: $lStack.label(lStack.getFromTop(k - \tau n + 1).label, val)$
43: $setVolatility(elem, k - \tau n + 1)$
44: $lStack.label(null, lStack.getFromTop(k +$
$\min\{|lowerRegion()|, popsInPending\} + 1)$
45: $lStack.topAvailableByLabel(p_i).canPopByLabel := false$
46: **if** $j == i$ and $op == push\_f$ **then**
47: $localFastPushesActive--$
48: **else**
49: **if** $op == pop\_f$ **then**
50: **if** $j == i$ **then** $localPopsExecuted{+}{+}$
51: **if** $val == lStack.getFromTop(1)$ **then**
52: $localPopsInvoked -= localPopsExecuted$
53: $localPopsExecuted := 0$
54: let $ret := val$
55: **else**
56: **if** $localPopsExecuted == l_s$ **then**
57: $localPopsInvoked -= localPopsExecuted$
58: $localPopsExecuted := 0$

```
59:                 let ret := lStack.peek()
60:             else
61:                 if j == i then
62:                     localPopsInvoked++
63:                     localPopsExecuted++
64:                 let ret := lStack.peekByLabel(p_j)
65:             if j == i then return ret
66:         ret.popped := true
67:         ret.unresolved := true
68:         if ¬ret.canPopByLabel and ret.label == p_i then
69:             lStack.peekByLabel(p_i).canPopByLabel := false
70:         if ret.higher ≠ null then resetVolatility(ret.higher)
71:         popsInPending−−
72:         if ret == lStack.getFromTop(1) then
73:             while lStack has unresolved elements do
74:                 let toRemove := the topmost unresolved element
75:                 if toRemove ∈ safeRegion() then
76:                     if volatileRegion() ∩ lStack.topSet(k − τn) ≠ ∅ then
77:                         let toRelabel := lStack.peekBySet(volatileRegion())
78:                         resetVolatility(toRelabel)
79:                     else let toRelabel := lStack.getFromTop(k − τn + 1)
80:                     lStack.label(j, toRelabel)
81:                 lStack.remove(toRemove)
82:                 if lStack.labeledSize() < k then
83:                     lStack.label(dummy, lStack.peekByLabel(null))
84: function SafeRegion
85:     return [topSet(k − τn) ∪ topSetBySet(popsInPending, lowerRegion())] \
            topSetBySet(popsInPending, volatileRegion())
86: function VolatileRegion
87:     return {elem ∈ lStack.topSet(k − τn) : elem.t_volatile > localTime − (2d + 2u + ϵ)}
88: function LowerRegion
89:     return {elem.lower : elem ∈ volatileRegion()}
90: function SetVolatility(elem, other)
91:     elem.t_volatile := localTime
92:     elem.lower := other
93:     elem.lower.higher := elem
94: function ResetVolatility(elem)
```

95: $elem.t_{volatile} := -\infty$
96: $elem.lower.higher := null$
97: $elem.lower := null$

A.2.1 Correctness

Let $ts(op)$ denote the timestamp associated with an operation instance op given as the first argument in line 3, 7, 14, or 16.

Construction 3. *Define the permutation π of operation instances in a complete, admissible run of Algorithm 4 as the order given by sorting by $ts(op)$ for each operation instance op.*

First, the following two lemmas have the same proofs as in Algorithm 3.

Lemma 32. *Each process running Algorithm 4 locally executes all $Pushes$ and $Pops$ in timestamp order.*

Lemma 33. *π respects the real-time order of non-overlapping operation instances.*

The lemmas up to Lemma 40 are the same as for Algorithm 2, and their proofs are identical.

Lemma 34. *There are at most τ_c mutually active $Pushes$ occurring in a single process and at most $\tau_c n$ mutually active $Pushes$ occurring over all processes at any given time.*

Lemma 35. *The $safeRegion()$ function given in Algorithm 4 returns a set of elements that are guaranteed to be safe.*

Definition 21. *The* uniformity invariant *is given by Definition 12.*

Lemma 36. *If an element $elem$ is the $lower$ element in some $(higher, lower)$-pair, then the value of $elem.higher$ is the other element; if not, $elem.higher$ is $null$. Likewise, if $elem$ is the $higher$ element in some $(higher, lower)$-pair, then the value of $elem.lower$ is the other element, and if not, $elem.lower$ is $null$.*

Lemma 37. *The* volatility invariant *holds, and states that in the local stack of each process,*

(i) any two elements that form a $(higher, lower)$*-pair share the same label;*

(ii) if an element $elem$ *is the topmost in the volatile region, then* $elem.lower$ *is currently the* $(k - \tau n + 1)$*th-topmost element, so that any element is the* m*th-topmost volatile element if and only if it is the* $higher$ *of a* $(higher, lower)$*-pair in which* $lower$ *is the* m*th-topmost element in* $lowerRegion()$*;*

(iii) the lower region is mutually exclusive with the topmost $k - \tau n$ *elements and connected; and*

(iv) member elements of a $(higher, lower)$*-pair cannot mutually exist in the safe region.*

Corollary 38. *The size of the safe region is always* $\min\{k - \tau n, lStack.size()\}$.

Corollary 39. *Incrementing or decrementing* $popsInPending$ *preserves the uniformity invariant.*

Remark 4. $popsInPending$ is incremented by 1 (line 21) every time the process receives a message sent from any Pop and is decremented by 1 (line 71) every time any Pop executes locally, which means this variable counts the number of active Pops from which this process received a message.

Definition 22. *The* labeling invariant *is given by Definition 16.*

Lemma 40. *Any element* $elem$ *returned by a* Pop *will not be relabeled with another process id unless* $elem.poppedState = executed$ *in all processes.*

Proof. The proof is identical to Lemma 10 except with one difference in Algorithm 4 being that not all Pops remove their returned element $elem$ at local execution, but they do set $elem.poppedState := executed$. □

Additionally, the following lemmas are the same as for Algorithm 3, and their proofs are identical.

Lemma 41. *In each process,* $localPopsInvoked \geq localPopsExecuted$.

Lemma 42. *Every* Pop *returns a unique element.*

Now we demonstrate that the linearization given by the algorithm is a legal sequence of operation instances according to Definition 6.

Theorem 43. *For any execution of Algorithm 4, the permutation* π *given by Definition 3 is legal by the specification of a Windowed* k*-Relaxed Stack. Thus, Algorithm 4 is a correct implementation of an Windowed* k*-Relaxed Stack.* □

Proof. We consider local execution by operation type, and show that for every sequence of operation instances invoked by the processes, the algorithm (a) generates return values such that π is legal, and also maintains (b) the labeling invariant, (c) the uniformity invariant, and (d) the headmost invariant.

Since every process locally executes operation instances in linearized order, elements whose $unresolved$ field is $true$ correspond to unresolved elements in Definition 6, because they have been returned by Pops which have since executed locally (line 67), and they are removed by locally executing Pops which remove the top element (lines 73-83).

Along with the fact that not every locally executing Pop removes elements but those that do perform the same procedure as in Algorithm 2 albeit multiple times, the above shows that the reasoning for (a), (b), and (c) is identical to the proof of Theorem 12, except for (a) under Pop in two cases:

- If a fast Pop, pop, returns $\perp$ then we still have $\lambda(\rho \cdot pop) = \lambda(\rho) + 1$. We re-introduce $localPopsInvoked$ and $localPopsExecuted$ from Algorithm 3 to ensure that $\lambda(\rho \cdot pop) < k$, and the proof that it does so is the same as before.
- Slow Pops behave similarly to and follow the same reasoning for legality as in the proof of Theorem 29.

Additionally, the reasoning for (d) is the same as that given in the proof of Theorem 29. □

A.2.2 Performance

The mechanism for $Push$ and hence its upper bound proofs are the same as they were in Algorithm 2.

Theorem 44. *The worst-case time complexity of $Push$ among all complete, admissible executions of Algorithm 4 is no more than* $\max\{2d + (1 - \tau_c)\epsilon, \epsilon\}$. □

Theorem 45. *The amortized time complexity of $Push$ among all complete, admissible executions of Algorithm 4 is no more than* $\max\{2d/\tau_c, \epsilon\}$. □

This is the first algorithm we encounter in which the headmost invariant has a slight impact on the amortized result.

Theorem 46. *The amortized time complexity of Pop among all complete, admissible executions of Algorithm 4 is no more than* $(2d - \epsilon)/\min\{l_s, l - \tau\} + \epsilon$ *if* $\tau < l$, *or* $2d$ *otherwise.* □

Proof. First we assume $\tau < l$, and consider the view of a single process p_i in which a slow Pop occurs, which is either when $localPopsInvoked$ has been incremented to l_s by the same number of fast Pops or when all the elements labeled p_i that are not the top element have been returned in a fast Pop and $lStack.size() \leq k - \tau n$. Considering the proof of Theorem 31, at least $l_s - 1$ fast Pops occur before a slow Pop in the former. In the latter, at least $l - \tau - 1$ fast Pops occur. Since each fast Pop takes ϵ time and the slow Pop takes $2d$ time, the average cost of each repetition of a pattern of $\min\{l_s - 1, l - \tau - 1\}$ fast Pops and one slow Pop is

$$\frac{\min\{l_s - 1, l - \tau - 1\}\epsilon + 2d}{\min\{l_s, l - \tau\}} = \frac{\min\{l_s, l - \tau\}\epsilon + 2d - \epsilon}{\min\{l_s, l - \tau\}} = \frac{2d - \epsilon}{\min\{l_s, l - \tau\}} + \epsilon.$$

The cost of any prefix of the infinite repetition of this pattern is bounded above by the maximum average cost of a single copy of this pattern, since prefixes ending with a slow Pop which ends a pattern have the highest average cost.

Otherwise, if $\tau = l$, then the size of the safe region is smaller than n, so there may be no elements labeled p_i. Then every Pop is slow, taking $2d$ time.

In either case, because this provides an upper bound on the cost of Pops at any process p_i, it is also an upper bound on the average cost of Pops at all processes; hence, it is the amortized cost. □

A.3 Restricted Relaxed Stacks

Algorithm 5 implements the Restricted k-Relaxed Stack (Definition 5), which should be compared side-by-side with both Algorithm 2 and Algorithm 4. The Restricted k-Relaxed Stack has the same upper bounds as the Windowed k-Relaxed Stack.

Because operation instances execute in linearized order, the requirement of Definition 5 is that only the topmost $k - \lambda(\rho)$ elements are legal to return, where ρ is the linearization of operation instances that have executed locally. To obtain the functionality of allowing only these elements to be returned, elements are no longer relabeled every time the local execution of a fast Pop removes an element, so each one reduces the number of labeled elements.

Furthermore, we employ the local variable $kMinusLateness$ to keep track of the quantity $k - \lambda(\rho)$. In almost all instances k was used in Algorithm 2, k is now replaced with $kMinusLateness$.

Thus, Algorithm 5 has weaker *uniformity invariant* since the size of the safe region may be smaller. More specifically, the first part still always holds, but the second part holds in fewer situations.

A slow Pop always removes the top element, which according to the definition must reset the lateness. Thus, the local execution of slow Pops labels elements until the size of the safe region is $\min\{k - \tau n, lStack.size()\}$ and the number of labeled elements is $\min\{k, lStack.size()\}$.

Algorithm 5 Pseudocode for each process p_i implementing a Restricted k-Relaxed Stack, where $k \geq n$.

1: **HandleEvent** PUSH(val)
2: **if** $localFastPushesActive < \tau_c - 1$ **then**
3: send $(push\_f, val, \langle localTime, i\rangle)$ to all
4: $setTimer(\epsilon, \langle push\_f, null, \langle localTime, i\rangle\rangle, respond)$
5: $localFastPushesActive{+}{+}$
6: **else**
7: send $(push\_s, val, \langle localTime, i\rangle)$ to all
8: $setTimer(\epsilon, \langle push\_s, 1, \langle localTime, i\rangle\rangle, respond)$
9: **HandleEvent** POP
10: **if** ($lStack.peekByLabel(p_i, safeRegion()) \neq \perp$ or $lStack.size() < kMinusLateness - \tau n$) and $localPopsInvoked < l_s$ **then**
11: $localPopsInvoked{+}{+}$
12: let $ret := lStack.peekByLabel(p_i, safeRegion())$
13: $ret.popped := true$
14: send $(pop\_f, ret, \langle localTime, i\rangle)$ to all
15: $setTimer(\epsilon, \langle pop\_f, ret, null\rangle, respond)$
16: **else** send $(pop\_s, null, \langle localTime, i\rangle)$ to all
17: **HandleEvent** RECEIVE (op, val, ts) FROM p_j
18: $Pending.insert(\langle op, val, ts\rangle)$
19: $setTimer(d + u, \langle op, val, ts\rangle, execute)$
20: **if** $op == pop\_f$ or $op == pop\_s$ **then**
21: $popsInPending{+}{+}$
22: **if** $op == pop\_f$ **then**
23: $val.popped := true$
24: $resetVolatility(val)$
25: **HandleEvent** EXPIRETIMER($\langle op, val, \langle *, j\rangle\rangle, respond$)
26: **if** $op == pop\_f$ **then return** val
27: **else if** $op == push\_s$ and $localFastPushesActive == \tau_c - 1$ **then**
28: $setTimer(\min\{\epsilon, 2d - (\tau_c - 1)\epsilon - val \cdot \epsilon\}, \langle push\_s, val + 1, \langle *, j\rangle\rangle, respond)$
29: **else return** ACK
30: **HandleEvent** EXPIRETIMER($\langle op, val, ts\rangle, execute$)
31: **while** $ts \geq Pending.min()$ **do**
32: $\langle op', val', ts'\rangle := Pending.extractMin()$
33: $executeLocally(op', val', ts')$
34: $cancelTimer(\langle op', val', ts'\rangle, execute)$
35: **function** EXECUTELOCALLY($op, val, \langle time, j\rangle$)

```
36:  if op == push_f or op == push_s then
37:      lStack.topAvailableByLabel(p_i).canPopByLabel := true
38:      lStack.push(val)
39:      if lStack.size() ≤ kMinusLateness − τn then
40:          lStack.label(argmin_{p_h} |lStack.setByLabel(p_h) ∩ safeRegion()|, val)
41:      else
42:          lStack.label(lStack.getFromTop(kMinusLateness − τn + 1).label, val)
43:          setVolatility(val, lStack.getFromTop(kMinusLateness − τn + 1))
44:          lStack.label(null, lStack.getFromTop(kMinusLateness +
                 min{|lowerRegion()|, popsInPending} + 1)
45:      lStack.topAvailableByLabel(p_i).canPopByLabel := false
46:      if j == i and op == push_f then
47:          localFastPushesActive−−
48:  else
49:      if op == pop_f then
50:          let hasTop := (val == lStack.getFromTop(1))
51:          if j == i then localPopsExecuted++
52:          if hasTop then
53:              localPopsInvoked −= localPopsExecuted
54:              localPopsExecuted := 0
55:          lStack.remove(val)
56:      else
57:          if localPopsExecuted == l_s then
58:              let hasTop := (lStack.peek() == lStack.getFromTop(1))
59:              localPopsInvoked −= localPopsExecuted
60:              localPopsExecuted := 0
61:              let ret := lStack.pop()
62:              if ret.label == p_i then
63:                  lStack.peekByLabel(p_i).canPopByLabel := false
64:          else
65:              let hasTop := false
66:              if j == i then
67:                  localPopsInvoked++
68:                  localPopsExecuted++
69:              let ret := lStack.popByLabel(p_j)
70:          if j == i then return ret
71:      if ret.higher ≠ null then resetVolatility(ret.higher)
72:      kMinusLateness−−
```

73: $popsInPending--$

74: **if** $hasTop$ **then**

75: **while** $kMinusLateness < k$ **do**

76: let $labelNew := argmin_{p_h} |lStack.setByLabel(p_h) \cap safeRegion()|$

77: **if** $volatileRegion() \cap lStack.topSet(kMinusLateness - \tau n) \neq \emptyset$ **then**

78: let $toRelabel := lStack.peekBySet(volatileRegion())$

79: $resetVolatility(toRelabel)$

80: **else** let $toRelabel := lStack.getFromTop(kMinusLateness - \tau n + 1)$

81: $lStack.label(labelNew, toRelabel)$

82: $kMinusLateness{+}{+}$

83: **while** $lStack.labeledSize() < \min\{k, lStack.size()\}$ **do**

84: $lStack.label(dummy, lStack.peekByLabel(null))$

85: **function** SAFEREGION

86: **return** $[topSet(kMinusLateness - \tau n) \cup topSetBySet(popsInPending, lowerRegion())] \setminus topSetBySet(popsInPending, volatileRegion())$

87: **function** VOLATILEREGION

88: **return** $\{elem \in lStack.topSet(k - \tau n) : elem.t_{volatile} > localTime - (2d + 2u + \epsilon)\}$

89: **function** LOWERREGION

90: **return** $\{elem.lower : elem \in volatileRegion()\}$

91: **function** SETVOLATILITY($elem, other$)

92: $elem.t_{volatile} := localTime$

93: $elem.lower := other$

94: $elem.lower.higher := elem$

95: **function** RESETVOLATILITY($elem$)

96: $elem.t_{volatile} := -\infty$

97: $elem.lower.higher := null$

98: $elem.lower := null$

A.3.1 Correctness

Let $ts(op)$ denote the timestamp associated with an operation instance op given as the first argument in line 3, 7, 14, or 16.

Construction 4. *Define the permutation π of operation instances in a complete, admissible run of Algorithm 5 as the order given by sorting by $ts(op)$ for each operation instance op.*

First, the following two lemmas have the same proofs as in Algorithm 3.

Lemma 47. *Each process running Algorithm 5 locally executes all $Pushes$ and $Pops$ in timestamp order.*

Lemma 48. *π respects the real-time order of non-overlapping operation instances.*

We proceed in the same manner as in proving the correctness of Algorithm 2, but a major difference is that k is now replaced with $kMinusLateness$. Otherwise, the proofs not explicitly stated are identical as before.

Lemma 49. *There are at most τ_c mutually active $Pushes$ occurring in a single process and at most $\tau_c n$ mutually active $Pushes$ occurring over all processes at any given time.*

Lemma 50. *The $safeRegion()$ function given in Algorithm 5 returns a set of elements that are guaranteed to be safe.*

Definition 23. *The* uniformity invariant *states that*

$$|lStack.setByLabel(p_i) \cap safeRegion()| \leq \left\lceil \frac{k}{n} \right\rceil - \tau \quad \forall i \in [0, n-1]$$

with the number of i such that the bound is reached being at most $n_a := k - (\lceil k/n \rceil - 1)n$, and when $lStack.size() \geq k - \tau n$ with $kMinusLateness = k$,

$$|lStack.setByLabel(p_i) \cap safeRegion()| - |lStack.setByLabel(p_j) \cap safeRegion()| \leq 1$$
$$\forall i, j \in [0, n-1],$$

Lemma 51. *If an element $elem$ is the $lower$ element in some $(higher, lower)$-pair, then the value of $elem.higher$ is the other element; if not, $elem.higher$ is $null$. Likewise, if $elem$ is the $higher$ element in some $(higher, lower)$-pair, then the value of $elem.lower$ is the other element, and if not, $elem.lower$ is $null$.*

While the statement of the volatility invariant is still similar as before, its proof has differences with that of Lemma 7.

Lemma 52. *The* volatility invariant *holds, and states that in the local stack of each process,*

(i) any two elements that form a $(higher, lower)$*-pair share the same label;*

(ii) if an element $elem$ *is the topmost in the volatile region, then* $elem.lower$ *is currently the* $(kMinusLateness - \tau n + 1)$*th-topmost element, so that any element is the* m*th-topmost volatile element if and only if it is the* $higher$ *of a* $(higher, lower)$*-pair in which* $lower$ *is the* m*th-topmost element in* $lowerRegion()$*;*

(iii) the lower region is mutually exclusive with the topmost $kMinusLateness - \tau n$ *elements and connected; and*

(iv) member elements of a $(higher, lower)$*-pair cannot mutually exist in the safe region.*

Proof. We refer to the proof of Lemma 7. The reasoning still holds, because although $kMinusLateness$ is not constant, locally executing fast Pops only decrease this quantity and move all the existing $lower$ elements of $(higher, lower)$-pairs closer to the top, so they maintain the constraints given the new $kMinusLateness$ value.

The local execution of a slow Pop, which labels up to k elements, marks all the elements in the topmost $kMinusLateness - \tau n$ non-volatile (lines 78-79), making them no longer part of some $(higher, lower)$-pair and hence vacuously satisfying the contraints. Volatile elements outside the topmost $kMinusLateness - \tau n$ are unchanged. □

Corollary 53. *The size of the safe region is always* $\min\{kMinusLateness - \tau n, lStack.size()\}$.

Corollary 54. *Incrementing or decrementing* $popsInPending$ *preserves the uniformity invariant.*

Remark 5. $popsInPending$ is incremented by 1 (line 21) every time the process receives a message sent from any Pop and is decremented by 1 (line 73) every time any Pop executes locally, which means this variable counts the number of active Pops from which this process received a message.

Definition 24. *The* labeling invariant *states that the labeled elements in the local stack are the topmost* $kMinusLateness + \min\{|lowerRegion()|, popsInPending\}$ *elements.*

Lemma 55. *Any element* $elem$ *returned by a* Pop *will not be relabeled with another process id.*

Proof. Only slow Pops relabel elements, and by the time a slow Pop does so at local execution, all other invoked Pops with earlier timestamps will have already executed locally, removing their returned elements. □

As before, the following lemmas are the same as for Algorithm 3, and their proofs are identical.

Lemma 56. *In each process,* $localPopsInvoked \geq localPopsExecuted$.

Lemma 57. *Every* Pop *returns a unique element.*

Now we demonstrate that the linearization given by the algorithm is a legal sequence of operation instances according to Definition 5.

Theorem 58. *For any execution of Algorithm 5, the permutation* π *given by Definition 4 is legal by the specification of a Restricted* k*-Relaxed Stack. Thus, Algorithm 5 is a correct implementation of a Restricted* k*-Relaxed Stack.* □

Proof. We consider local execution by operation type, and show that for every sequence of operation instances invoked by the processes, the algorithm (a) generates return values such that π is legal, and also maintains (b) the labeling invariant, (c) the uniformity invariant,

and (d) the headmost invariant. The invariants are demonstrated by induction on the local stack, with the base case being that they are vacuously satisfied when the stack is empty.

The reasoning for (a), (b), and (c) is for the most part identical to that found in the proof of Theorem 12 except with k replaced by $kMinusLateness$, and the reasoning for (d) is the same as that given in the proof of Theorem 29. We list only the differences below, letting ρ be the linearization of all the operation instances with earlier timestamps.

Pop:

(a) The reasoning for fast Pops is the same as in (a) under Pop in the proof of Theorem 12, except with k replaced by $kMinusLateness$ as expected, as well as $localPopsInvoked$ and $localPopsExecuted$ again introduced from Algorithms 3 and 4 to limit the number of fast Pops returning $\perp$, the proof that it does so being the same as before.

$kMinusLateness = k - \lambda(\rho)$ because locally executing Pops that do not return the top element (so that they are not headmost according to Definition 5) decrement $kMinusLateness$, and those that do (so that they are headmost) reset $kMinusLateness$ to k.

Even though $kMinusLateness$ is not constant, the local execution of a mutually active Pop decreases $kMinusLateness$ also removes an element from the $kMinusLateness$ topmost, so any other elements that were in the topmost $kMinusLateness$ are still legal to return.

The reasoning for slow Pops behaving legally is the same as in Algorithm 29.

(b) Each locally executing Pop that does not return the top element decreases $kMinusLateness$ simultaneously while removing the element, and each locally executing Pop that returns the top element relabels $k - kMinusLateness$ elements (lines 81-82) and resets $kMinusLateness = k$, maintaining the labeling invariant.

(c) If the Pop does not return the top element, the first part of the uniformity invariant

holds since no elements are relabeled, and the second part does not apply because $kMinusLateness < k$.

If it returns the top element, then it chooses the same new labels for elements (line 76) as done by $Push$es, so the proof for (c) under $Push$ in the proof of Theorem 12 applies here and implies that both the first and second uniformity invariants are satisfied.

□

A.3.2 Performance

The mechanism for $Push$ and hence its upper bound proofs are the same as they were in Algorithm 2.

Theorem 59. *The worst-case time complexity of $Push$ among all complete, admissible executions of Algorithm 5 is no more than* $\max\{2d + (1 - \tau_c)\epsilon, \epsilon\}$. □

Theorem 60. *The amortized time complexity of $Push$ among all complete, admissible executions of Algorithm 5 is no more than* $\max\{2d/\tau_c, \epsilon\}$. □

This is the first algorithm we encounter in which the headmost invariant has a slight impact on the amortized result.

Theorem 61. *The amortized time complexity of Pop among all complete, admissible executions of Algorithm 5 is no more than* $(2d - \epsilon)/\min\{l_s, l - \tau\} + \epsilon$ *if* $\tau < l$, *or* $2d$ *otherwise.* □

Proof. The proof is similar to that of Theorem 46, and gives the same time complexity. □

A.4 Stuttering Relaxed Queues

Algorithm 2 implements the Stuttering k-Relaxed Queue, which is defined in a corresponding manner to the Stuttering k-Relaxed Stack (Definition 7).

The algorithm bears similarity to the Lateness k-Relaxed Queue algorithm (Algorithm 3), as $localDequeuesInvoked$ and $localDequeuesExecuted$ limit the number of fast $Dequeue$s in each process since the local execution of the previous slow $Dequeue$ in the linearization. It is also in a manner like an opposite, as $Dequeue$s are only allowed to return the top element instead of any element in the local queue.

Only slow $Dequeue$s remove elements from the local queue.

To certify that $Dequeue$s do not return elements that have already been removed according to Definition 7, we prevent fast $Dequeue$s from returning certain elements by marking those elements *popped* if the fast $Dequeue$s' timestamps are later by a certain amount than the timestamp of a slow $Dequeue$ returning the same element.

Interestingly, this is the only algorithm that allows for $0 \leq k < n$, and if $k = 0$, it implements an unrelaxed Queue.

Algorithm 6 Pseudocode for each process p_i implementing a Stuttering k-Relaxed Queue, where $k \geq 0$.

```
1:  HandleEvent ENQUEUE(val)
2:      send (enq, val, ⟨localTime, i⟩) to all
3:      setTimer(ϵ, ⟨enq, null, ⟨localTime, i⟩⟩, respond)
4:  HandleEvent DEQUEUE
5:      if localDeqsInvoked < l_s and lQueue.peek() ≠ ⊥ then
6:          localDeqsInvoked++
7:          while lQueue.peek().t_popped + (d + ϵ) < localTime do
8:              lQueue.peek().popped := true
9:          let ret := lQueue.peek()
10:         setTimer(ϵ, ⟨deq_f, ret, null⟩, respond)
11:         send (deq_f, null, ⟨localTime, i⟩) to all
12:     else send (deq_s, null, ⟨localTime, i⟩) to all
13: HandleEvent RECEIVE (op, val, ⟨t, j⟩) FROM p_j
14:     Pending.insert(⟨op, val, ⟨t, j⟩⟩)
15:     setTimer(u + ϵ, ⟨op, val, ⟨t, j⟩⟩, execute)
16:     if op == deq_s then val.t_popped := t
17: HandleEvent EXPIRETIMER(⟨op, val, ts⟩, respond)
18:     if op == deq_f then return val
19:     else return ACK
```

```
20: HandleEvent ExpireTimer(⟨op, val, ts⟩, execute)
21:     while ts ≥ Pending.min() do
22:         ⟨op′, val′, ts′⟩ := Pending.extractMin()
23:         executeLocally(op′, val′, ts′)
24:         cancelTimer(⟨op′, val′, ts′⟩, execute)
25: function ExecuteLocally(op, val, ⟨∗, j⟩)
26:     if op == enq then lQueue.enq(val)
27:     else if op == deq_f then localDeqsExecuted++
28:     else
29:         localDeqsInvoked −= localDeqsExecuted
30:         localDeqsExecuted := 0
31:         let ret := lQueue.deq()
32:         if j == i then return ret
```

A.4.1 Correctness

Let $ts(op)$ denote the timestamp associated with an operation instance op given as the first argument in line 2, 10, or 12.

Construction 5. *Define the permutation π of operation instances in a complete, admissible run of Algorithm 6 as the order given by sorting by $ts(op)$ for each operation instance op.*

In Algorithm 6, the pending delay is $u + \epsilon$ (line 15), and so Lemma 1 holds.

Lemma 62. *Each process running Algorithm 6 locally executes all $Enqueue$s and $Dequeue$s in timestamp order.*

The proof of the following lemma is the same as before.

Lemma 63. *π respects the real-time order of non-overlapping operation instances.*

Lemma 64. *In each process, $localDeqsInvoked \geq localDeqsExecuted$.*

Proof. The proof is similar to that of Lemma 27. □

Theorem 65. *For any execution of Algorithm 6, the permutation π given by Definition 5 is legal by the specification of Stuttering k-Relaxed Queue. Thus, Algorithm 6 is a correct implementation of an Stuttering k-Relaxed Queue.* □

Proof. We consider local execution by operation type, and show that for every sequence of operation instances invoked by the processes, the algorithm generates return values such that π is legal.

$Enqueue$: As long as every value is unique, *(C2)* in the relaxed queue definition corresponding to Definition 7 is satisfied and hence any $Enqueue$ is legal.

$Dequeue$: Consider a $Dequeue$ instance, which we call deq. The stutter count, or $s(\rho \cdot deq)$ where ρ is the linearization of operation instances, is equivalent to the number of locally executing $Dequeue$s returning without removing the top element (line 40) after the last locally executing $Dequeue$ that returned a distinct element, since every process executes operation instances in linearized order.

The sum of $localDeqsExecuted$ among all processes gives $s(\rho \cdot deq)$ because each fast $Dequeue$ increments $localDeqsExecuted$ in some process (line 27) while each slow $Dequeue$, which removes the top element, resets $localDeqsExecuted$ in all processes (line 30).

From line 5, the sum of $localDeqsInvoked$ among all processes is at most $l_s n = (\lceil k/n \rceil - 1)n < k$. Hence, with the result of Lemma 64, $s(\rho \cdot deq) < k$ also.

Lastly, we must show that a $Dequeue$ cannot return an element higher in the local queue than one that has already been returned, or it would violate the condition that the element is not removed in ρ according to Definition 7.

A $Dequeue$ may only return a different element from any previous $Dequeue$s if it succeeds a slow $Dequeue$, deq_s, which removes the top element from the local queue in each process. deq_s sends a message to all processes to mark the t_{popped} field of its returned element with its own timestamp (line 16), which occurs within d real time in all processes. The element returned by deq_s will not be removed from the local queue in any process

before this occurs in all processes, because the earliest it may execute locally in any process is $(d-u)+(u+\epsilon) = d+\epsilon$.

Since local times differ by at most ϵ, marking elements $popped$ (line 8) if they were returned by slow $Dequeue$s timestamped at least $d+\epsilon$ earlier guarantees that all preceding $Dequeue$s return the same element, and all succeeding $Dequeue$s return the same element. □

A.4.2 Performance

Theorem 66. *The worst-case and amortized time complexities of $Enqueue$ among all complete, admissible executions of Algorithm 6 is no more than ϵ.* □

Theorem 67. *The worst-case time complexity of $Dequeue$ among all complete, admissible executions of Algorithm 6 is no more than $d+\epsilon$.* □

Theorem 68. *The amortized time complexity of $Dequeue$ among all heavily loaded, complete, admissible executions of Algorithm 6 is no more than $d/(l_s+1)+\epsilon$.* □

Proof. We consider the view of a single process p_i in which a slow $Dequeue$ occurs, which is when $count$ has been incremented to l_s by the same number of fast $Dequeue$s. Since each fast $Dequeue$ takes ϵ time and the slow $Dequeue$ takes $d+\epsilon$ time, the average cost of each repetition of a pattern of l_s fast $Dequeue$s and one slow $Dequeue$ is

$$\frac{l_s\epsilon + (d+\epsilon)}{l_s+1} = \frac{d+(l_s+1)\epsilon}{l_s+1} = \frac{d}{l_s+1} + \epsilon.$$

The cost of any prefix of the infinite repetition of this pattern is bounded above by the maximum average cost of a single copy of this pattern, since prefixes ending with a slow $Enqueue$ which ends a pattern have the highest average cost.

Because this provides an upper bound on the cost of $Enqueue$s at any process p_i, it is also an upper bound on the average cost of $Enqueue$s at all processes; hence, it is the amortized cost. □

Theorem 69. *The amortized time complexity of $Dequeue$ among all complete, admissible executions of Algorithm 6 is no more than $d + \epsilon$.* □

Proof. The worst case is when the execution has no invoked $Enqueue$s; then all $Dequeue$s are slow, and the amortized time complexity is the same as the worst-case time complexity, $d + \epsilon$. □

APPENDIX B

SAMPLE OF DERIVED ALGORITHMS

See Sections 4.1.1 and 4.1.2 for the procedures on deriving these and other similar algorithms from relaxed stack algorithms.

Algorithm 7 Pseudocode for each process p_i implementing an Out-of-Order k-Relaxed Priority Queue, where $k \geq n$.

1: **HandleEvent** INSERT(val)
2: **if** $localFastInsertsActive < \tau_c - 1$ **then**
3: send $(insert\_f, val, \langle localTime, i\rangle)$ to all
4: $setTimer(\epsilon, \langle insert\_f, val, \langle localTime, i\rangle\rangle, respond)$
5: $localFastInsertsActive{+}{+}$
6: **else**
7: send $(insert\_s, val, \langle localTime, i\rangle)$ to all
8: $setTimer(\epsilon, \langle insert\_s, null, \langle localTime, i\rangle\rangle, respond)$
9: **HandleEvent** EXTRACTMAX
10: **if** $lPQ.peekByLabel(p_i, safeRegion()) \neq \perp$ or $lPQ.size() < k - \tau n$ **then**
11: let $ret := lPQ.peekByLabel(p_i, safeRegion())$
12: $ret.extracted := true$
13: send $(extractMax\_f, ret, \langle localTime, i\rangle)$ to all
14: $setTimer(\epsilon, \langle extractMax\_f, ret, null\rangle, respond)$
15: **else**
16: send $(extractMax\_s, null, \langle localTime, i\rangle)$ to all
17: $setTimer(\epsilon, \langle extractMax\_s, 1, \langle localTime, i\rangle\rangle, respond)$
18: **HandleEvent** RECEIVE (op, val, ts) FROM p_j
19: $Pending.insert(\langle op, val, ts\rangle)$
20: $setTimer(d + u, \langle op, val, ts\rangle, execute)$
21: **if** $op == extractMax\_f$ or $op == extractMax\_s$ **then**
22: $extractMaxesInPending{+}{+}$
23: **if** $op == extractMax\_f$ **then**
24: $resetVolatility(val)$
25: **HandleEvent** EXPIRETIMER($\langle op, val, \langle *, j\rangle\rangle, respond$)
26: **if** $op == extractMax\_f$ **then return** val
27: **else if** $op == extractMax\_s$ **then**

28: $flushReservedQueueFront(j)$
29: **if** $reservedQueues[j].peek \neq \perp$ **then**
30: $reservedQueues[j].peek().extracted := true$
31: **return** $reservedQueues[j].peek()$
32: **else if** $val \cdot \epsilon - [2d - (l - \tau)\epsilon] < \epsilon$ **then**
33: $setTimer(\min\{\epsilon, 2d - (l - \tau)\epsilon - val \cdot \epsilon\}, \langle pop\_s, val + 1, \langle *, j\rangle\rangle, respond)$
34: **else return** $\perp$
35: **else if** $op == insert\_s$ and $localFastInsertsActive == \tau_c - 1$ **then**
36: $setTimer(\min\{\epsilon, 2d - (\tau_c - 1)\epsilon - val \cdot \epsilon\}, \langle push\_s, val + 1, \langle *, j\rangle\rangle, respond)$
37: **else return** ACK
38: **HandleEvent** EXPIRETIMER($\langle op, val, ts\rangle, execute$)
39: **while** $ts \geq Pending.min()$ **do**
40: $\langle op', val', ts'\rangle := Pending.extractMin()$
41: $executeLocally(op', val', ts')$
42: $cancelTimer(\langle op', val', ts'\rangle, execute)$
43: **function** EXECUTELOCALLY($op, val, \langle time, j\rangle$)
44: **if** $op == insert\_f$ or $op == insert\_s$ **then**
45: $lPQ.insert(val)$
46: **if** $lPQ.size() \leq k - \tau n$ **then**
47: $lPQ.label(\text{argmin}_{p_h} |lPQ.setByLabel(p_h) \cap safeRegion()|, val)$
48: **else if** $val \in topSet(lPQ.labeledSize())$ **then**
49: **if** $\exists elem \in safeRegion()$ s.t. $P(elem) < P(val)$ **then**
50: $lPQ.label(lPQ.getFromTop(k - \tau n + 1).label, val)$
51: $setVolatility(elem, k - \tau n + 1)$
52: **else** $lPQ.label(dummy, val)$
53: $lPQ.label(null, lPQ.getFromTop(k +$
$\min\{|lowerRegion()|, popsInPending\} + 1)$
54: **if** $j == i$ and $op == insert\_f$ **then** $localFastInsertsActive--$
55: **else**
56: **if** $j \neq i$ **then** $flushReservedQueueFront(j)$
57: **if** $op == extractMax\_f$ **then** let $toRemove := val$
58: **else** let $toRemove := reservedQueues[j].dequeue()$
59: **if** $toRemove.higher \neq null$ **then** $resetVolatility(toRemove.higher)$
60: $extractMaxesInPending--$
61: **if** $toRemove \in safeRegion()$ **then**
62: **if** $volatileRegion() \cap lPQ.topSet(k - \tau n) \neq \emptyset$ **then**
63: let $toRelabel := lPQ.latestBySet(volatileRegion())$
64: $resetVolatility(toRelabel)$

65: **else** let $toRelabel := lPQ.getFromTop(k - \tau n + 1)$

66: $lPQ.label(j, toRelabel)$

67: $reservedQueues[j].enqueue(toRelabel)$

68: $lPQ.remove(toRemove)$

69: **if** $lPQ.labeledSize() < k$ **then** $lPQ.label(dummy, lPQ.peekByLabel(null))$

70: **function** SAFEREGION

71: **return** $[topSet(k - \tau n) \cup topSetBySet(extractMaxesInPending, lowerRegion())] \setminus latestSetBySet(extractMaxesInPending, volatileRegion())$

72: **function** VOLATILEREGION

73: **return** $\{elem \in lPQ.topSet(k - \tau n) : elem.t_{volatile} > localTime - (2d + 2u + \epsilon)\}$

74: **function** LOWERREGION

75: **return** $\{elem.lower : elem \in volatileRegion()\}$

76: **function** SETVOLATILITY($elem, other$)

77: $elem.t_{volatile} := localTime$

78: $elem.lower := other$

79: $elem.lower.higher := elem$

80: **function** RESETVOLATILITY($elem$)

81: $elem.t_{volatile} := -\infty$

82: $elem.lower.higher := null$

83: $elem.lower := null$

84: **function** FLUSHRESERVEDQUEUEFRONT(j)

85: let $front := reservedQueues[j].peek()$

86: **while** $\neg lPQ.contains(front)$ or $front.extracted$ or $front.label \neq p_j$ **do**

87: $reservedQueues[j].dequeue()$

88: let $front := reservedQueues[j].peek()$

Algorithm 8 Pseudocode for each process p_i implementing an Out-of-Order k-Relaxed Queue, where $k \geq n$.

1: **HandleEvent** ENQUEUE(val)

2: send $(enq, val, \langle localTime, i\rangle)$ to all

3: $setTimer(\epsilon, \langle enq, null, \langle localTime, i\rangle\rangle, respond)$

4: **HandleEvent** DEQUEUE

5: **if** $lQueue.peekByLabel(p_i) \neq \perp$ **then**

6: let $ret := lQueue.peekByLabel(p_i)$

7: $ret.dequeued := true$

8: send $(deq\_f, null, \langle localTime, i\rangle)$ to all

9: $setTimer(\epsilon, \langle deq\_f, ret, null\rangle, respond)$

10: **else**

11: send $(deq\_s, null, \langle localTime, i\rangle)$ to all

12: $setTimer(\epsilon, \langle deq\_s, 1, \langle localTime, i\rangle\rangle, respond)$

13: **HandleEvent** RECEIVE (op, val, ts) FROM p_j

14: $Pending.insert(\langle op, val, ts\rangle)$

15: $setTimer(u + \epsilon, \langle op, val, ts\rangle, execute)$

16: **HandleEvent** EXPIRETIMER($\langle op, val, \langle *, j\rangle\rangle, respond$)

17: **if** $op == deq\_f$ **then return** val

18: **else if** $op == deq\_s$ **then**

19: **if** $lQueue.peekByLabel(p_j) \neq \perp$ **then**

20: let $ret := lQueue.peekByLabel(p_j)$

21: $ret.dequeued := true$

22: **return** ret

23: **else if** $val \cdot \epsilon - [(d + \epsilon) - l\epsilon] < \epsilon$ **then**

24: $setTimer(\min\{\epsilon, (d + \epsilon) - l\epsilon - val \cdot \epsilon\}, \langle deq\_s, val + 1, \langle *, j\rangle\rangle, respond)$

25: **else**

26: $setTimer(d + \epsilon - val \cdot \epsilon\}, \langle deq\_s2, null, \langle *, j\rangle\rangle, respond)$

27: **else if** $op == deq\_s2$ **then**

28: **return** $lQueue.deqByLabel(p_j)$

29: **else return** ACK

30: **HandleEvent** EXPIRETIMER($\langle op, val, ts\rangle, execute$)

31: **while** $ts \geq Pending.min()$ **do**

32: $\langle op', val', ts'\rangle := Pending.extractMin()$

33: $executeLocally(op', val', ts')$

34: $cancelTimer(\langle op', val', ts'\rangle, execute)$

35: **function** EXECUTELOCALLY($op, val, \langle *, j\rangle$)

36: **if** $op == enq$ **then**

37: $lQueue.enq(val)$

38: **if** $lQueue.size() \leq k$ **then**

39: $lQueue.label(\text{argmin}_{p_h} |lQueue.setByLabel(p_h)|, val)$

40: **else**

41: $lQueue.deqByLabel(p_j)$

42: **if** $lQueue.size() \geq k$ **then**

43: $lQueue.label(\text{argmin}_{p_h} |lQueue.setByLabel(p_h)|, lQueue.peekByLabel(null))$

Algorithm 9 Pseudocode for each process p_i implementing a Lateness k-Relaxed Queue, where $k \geq n$.

```
1: HandleEvent ENQUEUE(val)
2:     send (enq, val, ⟨localTime, i⟩) to all
3:     setTimer(ϵ, ⟨enq, null, ⟨localTime, i⟩⟩, respond)
4: HandleEvent DEQUEUE
5:     if localDeqsInvoked < l_s then
6:         localDeqsInvoked++
7:         let ret := lQueue.peekByLabel(p_i)
8:         ret.dequeued := true
9:         send (deq_f, ret, ⟨localTime, i⟩) to all
10:        setTimer(ϵ, ⟨deq_f, ret, null⟩, respond)
11:    else send (deq_s, null, ⟨localTime, i⟩) to all
12: HandleEvent RECEIVE (op, val, ts) FROM p_j
13:    Pending.insert(⟨op, val, ts⟩)
14:    setTimer(2u, ⟨op, val, ts⟩, execute)
15:    if op == deq_f then val.dequeued := true
16: HandleEvent EXPIRETIMER(⟨op, val, ts⟩, respond)
17:    if op == deq_f then return val
18:    else return ACK
19: HandleEvent EXPIRETIMER(⟨op, val, ts⟩, execute)
20:    while ts ≥ Pending.min() do
21:        ⟨op', val', ts'⟩ := Pending.extractMin()
22:        executeLocally(op', val', ts')
23:        cancelTimer(⟨op', val', ts'⟩, execute)
24: function EXECUTELOCALLY(op, val, ⟨∗, j⟩)
25:    if op == enq then
26:        lQueue.topAvailableByLabel(p_i).canDeqByLabel := true
27:        lQueue.enq(val)
28:        lQueue.label(argmin_{p_h} |lQueue.setByLabel(p_h)|, val)
29:        lQueue.topAvailableByLabel(p_i).canDeqByLabel := false
30:    else
31:        if op == deq_f then
32:            if j == i then localDeqsExecuted++
33:            if val == lQueue.getFromTop(1) then
34:                localDeqsInvoked −= localDeqsExecuted
35:                localDeqsExecuted := 0
36:            lQueue.remove(val)
37:        else
38:            if localDeqsExecuted == l_s then
```

```
39:                 localDeqsInvoked −= localDeqsExecuted
40:                 localDeqsExecuted := 0
41:                 let ret := lQueue.deq()
42:                 if ret.label == p_i then
43:                     lQueue.peekByLabel(p_i).canDeqByLabel := false
44:             else
45:                 if j == i then
46:                     localDeqsInvoked++
47:                     localDeqsExecuted++
48:                 let ret := lQueue.deqByLabel(p_j)
49:             if j == i then return ret
```

Algorithm 10 Pseudocode for each process p_i implementing a Restricted/Windowed k-Relaxed Queue, where $k \geq n$.

```
 1: HandleEvent ENQUEUE(val)
 2:     send (enq, val, ⟨localTime, i⟩) to all
 3:     setTimer(ϵ, ⟨enq, null, ⟨localTime, i⟩⟩, respond)
 4: HandleEvent DEQUEUE
 5:     if lQueue.peekByLabel(p_i) ≠ ⊥ then
 6:         let ret := lQueue.peekByLabel(p_i)
 7:         ret.dequeued := true
 8:         send (deq_f, null, ⟨localTime, i⟩) to all
 9:         setTimer(ϵ, ⟨deq_f, ret, null⟩, respond)
10:     else send (deq_s, null, ⟨localTime, i⟩) to all
11: HandleEvent RECEIVE (op, val, ts) FROM p_j
12:     Pending.insert(⟨op, val, ts⟩)
13:     setTimer(2u, ⟨op, val, ts⟩, execute)
14:     if op == deq_f then val.dequeued := true
15: HandleEvent EXPIRETIMER(⟨op, val, ⟨∗, j⟩⟩, respond)
16:     if op == deq_f then return val
17:     else return ACK
18: HandleEvent EXPIRETIMER(⟨op, val, ts⟩, execute)
19:     while ts ≥ Pending.min() do
20:         ⟨op′, val′, ts′⟩ := Pending.extractMin()
21:         executeLocally(op′, val′, ts′)
22:         cancelTimer(⟨op′, val′, ts′⟩, execute)
23: function EXECUTELOCALLY(op, val, ⟨∗, j⟩)
```

24: **if** $op == enq$ **then**
25: $lQueue.topAvailableByLabel(p_i).canDeqByLabel := true$
26: $lQueue.enq(val)$
27: **if** $lQueue.size() \leq k$ **then** $lQueue.label(\text{argmin}_{p_h} |lQueue.setByLabel(p_h)|, val)$
28: $lQueue.topAvailableByLabel(p_i).canDeqByLabel := false$
29: **else**
30: **if** $op == deq\_f$ **then**
31: **if** $j == i$ **then** $localDeqsExecuted{+}{+}$
32: **if** $val == lQueue.getFromTop(1)$ **then**
33: $localDeqsInvoked -= localDeqsExecuted$
34: $localDeqsExecuted := 0$
35: let $ret := val$
36: **else**
37: **if** $localDeqsExecuted == l_s$ **then**
38: $localDeqsInvoked -= localDeqsExecuted$
39: $localDeqsExecuted := 0$
40: let $ret := lQueue.peek()$
41: **else**
42: **if** $j == i$ **then**
43: $localDeqsInvoked{+}{+}$
44: $localDeqsExecuted{+}{+}$
45: let $ret := lQueue.peekByLabel(p_j)$
46: **if** $j == i$ **then return** ret
47: $ret.dequeued := true$
48: $ret.unresolved := true$
49: **if** $\neg ret.canDeqByLabel$ and $ret.label == p_i$ **then**
50: $lQueue.peekByLabel(p_i).canDeqByLabel := false$
51: **if** $ret == lQueue.getFromTop(1)$ **then**
52: **while** $lQueue$ has $unresolved$ elements **do**
53: $lQueue.remove($the topmost $unresolved$ element$)$
54: **if** $lQueue.size() \geq k$ **then**
55: $lQueue.label(\text{argmin}_{p_h} |lQueue.setByLabel(p_h)|, lQueue.peekByLabel(null))$

REFERENCES

[1] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *Acm transactions on programming languages and systems*, vol. 12, no. 3, pp. 463–492, 1990.

[2] J. Wang, E. Talmage, H. Lee, and J. L. Welch, "Improved time bounds for linearizable implementations of abstract data types," in *International parallel and distributed processing symposium (ipdps)*, 2014.

[3] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. T. Vechev, "Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated," in *Principles of programming languages (popl)*, 2011.

[4] R. J. Lipton and J. Sandberg, "Pram: A scalable shared memory," Princeton University, Tech. Rep., Sep. 1988.

[5] H. Attiya and J. L. Welch, "Sequential consistency versus linearizability," vol. 12, no. 12, pp. 91–122, May 1994.

[6] Y. Afek, G. Korland, and E. Yanovsky, "Quasi-linearizability: Relaxed consistency for improved concurrency," in *14th international conference on principles of distributed systems (opodis)*, Dec. 2010.

[7] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," vol. 28, no. 9, pp. 690–691, Sep. 1979.

[8] W. Vogels, "Eventually consistent," vol. 52, no. 1, pp. 40–44, 2009.

[9] J. Aspens, M. Herlihy, and N. Shavit, "Counting networks," vol. 41, no. 5, pp. 1020–1048, 1994.

[10] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova, "Quantitative relaxation of concurrent data structures," in *Principles of programming languages (popl)*, 2013.

[11] E. Talmage and J. L. Welch, "Improving average performance by relaxing distributed data structures," in *International symposium on distributed computing (disc)*, 2014.

[12] J. Lundelius and N. A. Lynch, "An upper and lower bound for clock synchronization," in *Information and control*, vol. 62, 1984, pp. 190–204.

[13] M. Wimmer, J. Gruber, J. L. Träff, and P. Tsigas, "The lock-free k-lsm relaxed priority queue," in *Proceedings of the 20th acm sigplan symposium on principles and practice of parallel programming (ppopp)*, Feb. 2015.

[14] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit, "The spraylist: A scalable relaxed priority queue," in *Proceedings of the 20th acm sigplan symposium on principles and practice of parallel programming (ppopp)*, Feb. 2015.